**Introduction: Complex Data Structures: 39chapter**

**Hash Maps**

Moving on from the linear data structures, now it’s time to focus on hash maps! Hash maps map keys to their related values, and are one of the most efficient data structures when it comes to retrieving stored data. This is because the key associated with every value added allows for faster retrieval later on. When you come across a coding problem that requires you to store and retrieve data, keep in mind that hash maps will often be the most efficient data structure for that scenario.

Hash map: A key-value store that uses an array and a hashing function to save and retrieve values.

Key: The identifier given to a value for later retrieval.

Hash function: A function that takes some input and returns a number.

Compression function: A function that transforms its inputs into some smaller range of possible outputs.

Recipe for saving to a hash table:

- Take the key and plug it into the hash function, getting the hash code.

- Modulo that hash code by the length of the underlying array, getting an array index.

- Check if the array at that index is empty, if so, save the value (and the key) there.

- If the array is full at that index continue to the next possible position depending on your collision strategy.

Recipe for retrieving from a hash table:

- Take the key and plug it into the hash function, getting the hash code.

- Modulo that hash code by the length of the underlying array, getting an array index.

- Check if the array at that index has contents, if so, check the key saved there.

- If the key matches the one you're looking for, return the value.

- If the keys don't match, continue to the next position depending on your collision strategy.

**Tables**

A data structure’s main utility is allowing for data to be represented in a way that resembles the way people will use that data.

**Maps**

Being a *map* means relating two pieces of information, but a map also has one further requirement. Let’s consider the following table:

| **Musician** | **State of Birth** |
| --- | --- |
| Miles Davis | Illinois |
| John Coltrane | North Carolina |
| Duke Ellington | Ohio |

### Hash Maps

Hash maps are a common data structure used to store key-value pairs for efficient retrieval. A value stored in a hash map is retrieved using the key under which it was stored.

# `states` is a Hash Map with state abbreviation keys and state name values.  
   
states = {  
  'TN': "Tennessee",  
  'CA': "California",  
  'NY': "New York",  
  'FL': "Florida"  
}  
   
west\_coast\_state = states['CA']

### Hash function

Hash map data structures use a hash function, which turns a key into an index within an underlying array. The hash function can be used to access an index when inserting a value or retrieving a value from a hash map.

### Hash map underlying data structure

Hash maps are built on top of an underlying array data structure using an indexing system.

Each index in the array can store one key-value pair. If the hash map is implemented using chaining for collision resolution, each index can store another data structure such as a linked list, which stores all values for multiple keys that hash to the same index.

### hash map only one value

Each Hash Map key can be paired with only one value. However, different keys can be paired with the same value.

#This is a valid Hash Map where 2 keys share the same value  
correct\_hash\_map = {  
  "a" : 1,  
  "b" : 3,  
  "c" : 1  
}  
   
#This Hash Map is INVALID since a key cannot have more than 1 value  
incorrect\_hash\_map = {  
  "a" : 1,  
  "a" : 3,  
  "b" : 2  
}

**Collisions**

The first strategy we’re going to learn about is called separate chaining. The separate chaining strategy avoids collisions by updating the underlying data structure. Instead of an array of values that are mapped to by hashes, it could be an array of linked lists!

**Separate Chaining**

A hash map with a linked list separate chaining strategy follows a similar flow to the hash maps that have been described so far

**Saving Keys**

A hash collision resolution strategy like separate chaining involves assigning two keys with the same hash to different parts of the underlying data structure.

**Open Addressing: Linear Probing**

Another popular hash collision strategy is called *open addressing*. In open addressing we stick to the array as our underlying data structure, but we continue looking for a new index to save our data if the first result of our hash function has a different key’s data.

A common open method of open addressing is called *probing*. Probing means continuing to find new array indices in a fixed sequence until an empty index is found.

Suppose we want to associate famous horses with their owners. We want our first key, “Bucephalus”, to store our first value, “Alexander the Great”. Our hash function returns an array index 3 and so we save “Alexander the Great”, along with our key “Bucephalus”, into the array at index 3.

After that, we want to store “Seabiscuit”s owner “Charles Howard”. Unfortunately “Seabiscuit” also has a hash value of 3. Our probing method adds one to the hash value and tells us to continue looking at index 4. Since index 4 is open we store “Charles Howard” into the array at index 4. Because “Seabiscuit” has a hash of 3 but “Charles Howard” is located at index 4, we must also save “Seabiscuit” into the array at that index.

When we attempt to look up “Seabiscuit” in our Horse Owner’s Hash Map, we first check the array at index 3. Upon noticing that our key (Seabiscuit) is different from the key sitting in index 3 (Bucephalus), we realize that this can’t be the value we were looking for at all. Only by continuing to the next index do we check the key and notice that at index 4 our key matches the key saved into the index 4 bucket. Realizing that index 4 has the key “Seabiscuit” means we can retrieve the information at that location, Seabiscuit’s owner’s name: Charles Howard.

**HASH MAPS: CONCEPTUAL**

**Other Open Addressing Techniques**

There are more sophisticated ways to find the next address after a hash collision, although anything too calculation-intensive would negatively affect a hash table’s performance. Linear probing systems, for instance, could jump by five steps instead of one step.

In a quadratic probing open addressing system, we add increasingly large numbers to the hash code. At the first collision we just add 1, but if the hash collides there too we add 4 ,and the third time we add 9. Having a probe sequence change over time like this avoids clustering.

Clustering is what happens when a single hash collision causes additional hash collisions. Imagine a hash collision triggers a linear probing sequence to assigns a value to the next hash bucket over. Any key that would hash to this “next bucket” will now collide with a key that, in a sense, doesn’t belong to that bucket anyway.

As a result the new key needs to be assigned to the next, next bucket over. This propagates the problem because now there are two hash buckets taken up by key-value pairs that were assigned as a result of a hash collision, displacing further pairs of information we might want to save to the table.

A hash map is:

* Built on top of an array using a special indexing system.
* A key-value storage with fast assignments and lookup.
* A table that represents a map from a set of keys to a set of values.

Hash maps accomplish all this by using a hash function, which turns a key into an index into the underlying array.

A hash collision is when a hash function returns the same index for two different keys.

There are different hash collision strategies. Two important ones are separate chaining, where each array index points to a different data structure, and open addressing, where a collision triggers a probing sequence to find where to store the value for a given key.

**Intro to Hash Maps**

*Hash maps* are data structures that serve as efficient key-value stores. They are capable of assigning and retrieving data in the fastest way possible. This is because the underlying data structure that hash maps use is an array.

A value is stored at an array index determined by plugging the key into a hash function. Because we always know exactly where to find values in a hash map, we have constant access to any of the values it contains.

The hash map class we will be completing is stored in the **HashMap.js** file. Look over the code you’ve been given, the constructor method.

Hash maps are based on arrays, so whenever we create a new hash map instance, we’ll create an array filled with null values that can be replaced with key-value pairs later.

Checkpoint

Answer: HashMap.js

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

}

module.exports = HashMap;

**Hashing**

The *hashing function* is the secret to efficiently storing and retrieving values in a hash map. A hashing function takes a key as input and returns an index within the hash map’s underlying array.

This function is said to be *deterministic*. That means the hashing function must always return the same index when given the same key.

The hashing function should follow this logic:

declare hashCode variable with value of 0

for each character in the key

add the sum of the current character code value and hashCode to hashCode

return hashCode

Adding the sum of hashCode and the character code to the hashCode again creates a deterministic and also non-reversible implementation of a hashing function. This avoids generating a duplicate index if keys have the same characters in different orders, such as bat and tab.

### Instructions

**1.**

Create a HashMap method, .hash(), with key as a parameter. This method will take a string and use it to generate an index in the hash map’s internal array.

Checkpoint 2 Passed

**2.**

To generate an index for each key-value pair, we’ll calculate a number based on the characters in the input string. Declare a variable whose value can be changed within .hash() called hashCode. Assign it an initial value of 0.

This variable will keep a running total of character codes.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

After declaring hashCode create a for loop that loops over each character in key.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Inside of the for loop convert each character in key to an integer using the JavaScript string method .charCodeAt().

This method only works on strings and converts a character at a specific index into an integer between 0 and 65535. This integer represents the equivalent [Unicode](https://developer.mozilla.org/en-US/docs/Glossary/Unicode) value of that character.

To use .charCodeAt() call it on a string with the index of the character you want the character code of:

// The code below will return the character code of 'H'  
'Hello world!'.charCodeAt(0) // => 72

Add the result of calling .charCodeAt() on the current character of key and hashCode to the hashCode variable.

Outside of the for loop, return the finished hashCode.

Answer:

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode;

  }

}

module.exports = HashMap;

**Compression**

The current hashing function will return a wide range of integers — some of which are not indices of the hash map array. To fix this, we need to use compression.

Compression means taking some input and returning an output only within a specific range.

In our hash map implementation, we’re going to have our hashing function handle compression in addition to hashing. This means we’ll add an additional line of code to compress the hashCode before we return it.

The updated .hash() should follow these steps:

initialize hashCode variable to 0

for each character in the key

add the character code and hashCode to hashCode

return compressed hashCode

### Instructions

**1.**

Currently, our .hash() method is generating an integer representing an index but it’s not guaranteed that this index will be within the bounds of the hash map’s array.

To do this, we’ll use modular arithmetic. Because modular arithmetic prevents a value from growing larger than some limit, it’s a common solution when we want a value to “wrap around”.

After the for loop in the .hash() of HashMap, compress the value stored in hashCode by using modular arithmetic to return the remainder of dividing hashCode by the length of the hash map.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check your work. Save a new HashMap instance with a size of 3, in a constant myHashMap and use the new .hash() to log the result of hashing the key 'id'. Hash and log 'id' again. Are the logged values the same or are they different?

Checkpoint 3 Passed

answer:

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

}

module.exports = HashMap;

const myHashMap = new HashMap(3);

console.log(myHashMap.hash('id'));

console.log(myHashMap.hash('id'));

**Assign**

We now have everything we need to find a place in the hash map array to store a value. The only thing left to do is assign the value to the index we generated. A method, .assign() will handle the logic needed to take in a key-value pair and store the value at a particular index.

A general outline of how .assign() will work is this:

store the hashed key in a variable arrayIndex

assign the value to the element at arrayIndex in the hash map

### Instructions

**1.**

Declare a HashMap method called .assign() with the parameters key and value.

Checkpoint 2 Passed

**2.**

Declare a constant called arrayIndex with the value of the hashed and compressed key.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Assign the value to the element at the index you derived from hashing, arrayIndex.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Check your work. At the bottom of the **HashMap.js** file store a new instance of HashMap with a size of 3 in a constant named employees. Assign employees the key-value pair '34-567' and 'Mara', then log the hash map.

Checkpoint 5 Passed

answer:

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

}

module.exports = HashMap;

const employees = new HashMap(3);

employees.assign('34-567', 'Mara');

console.log(employees.hashmap);

**Retrieve**

To be a fully functional hash map, we have to be able to retrieve the values we are storing. To implement retrieval for our hash map we’ll create a new HashMap method, .retrieve().

This method will make use of .hash()‘s deterministic nature to find the value we’re looking for in the hash map.

### Instructions

**1.**

Define a method .retrieve() for HashMap. It should have one parameter, key, the key of the value we want to retrieve.

Checkpoint 2 Passed

**2.**

.retrieve() should calculate the array index in the same way .assign() does and then retrieve the value at that index.

Inside of .retrieve() declare a constant arrayIndex with the value of the hashed key. Use the HashMap method that takes a key and returns an index in the hash map’s array.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Return the value stored at arrayIndex.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Check your work. At the bottom of the **HashMap.js** file declare a new constant glossary that stores a hashmap with a size of 3.

* Add a new key of: 'semordnilap'
* With a value of: 'Words that form different words when reversed'

Log the result of retrieving 'semordnilap' from your glossary.

Answer:

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    return this.hashmap[arrayIndex];

  }

}

module.exports = HashMap;

const glossary = new HashMap(3);

glossary.assign('semordnilap', 'Words that form different words when reversed');

console.log(glossary.retrieve('semordnilap'));

**Collisions**

We have a hash map implementation, but what happens when two different keys generate the same index? Run the code in **collision.js** to see a collision in action.

Instead of returning 'marsh plant' and 'forest animal' we retrieve 'forest animal' twice. This is because both key-value pairs are assigned to the same index 0 and the first value, 'marsh plants' was overwritten.

When two different keys resolve to the same array index this is called a collision. In our current implementation, all keys that resolve to the same index are treated as if they are the same key. This is a problem because they will overwrite one another’s values.

### Instructions

**1.**

Run the code in the text editor to see the result of a collision between two keys.

Checkpoint 2 Passed

Answer: collision.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size);

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    this.hashmap[arrayIndex] = value;

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    return this.hashmap[arrayIndex];

  }

}

module.exports = HashMap;

const parkInventory = new HashMap(2);

parkInventory.assign('reed', 'marsh plant');

parkInventory.assign('deer', 'forest animal');

console.log(parkInventory.retrieve('reed'));

console.log(parkInventory.retrieve('deer'));

LinkedLsit.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Collisions: Assigning**

Our first step in implementing a collision strategy is updating our constructor and .assign() method to use linked lists and nodes inside the hashmap array. This will allow us to store multiple values at the same index by adding new nodes to a linked list instead of overwriting a single value. This strategy of handling collisions is called separate chaining.

A collision-proof .assign() should look like this to start:

store the hashed key in a variable arrayIndex

store linked list at arrayIndex in a variable linkedList

if linked list is empty

add the key-value pair to the linked list as a node

We’ll be using the LinkedList and Node classes found in the **LinkedList.js** and **Node.js** files to implement our collision-proof HashMap class. The only file you will be working in for this exercise is **HashMap**.

### Instructions

**1.**

Instead of an empty array, new hash maps will have an internal array filled with empty linked lists.

Tab to **HashMap.js**, and replace the code in the constructor method with the following:

this.hashmap = new Array(size)  
     .fill(null)  
     .map(() => new LinkedList());

Note the Array methods .fill() and .map() that are used to fill the array with placeholder values of null which are then replaced by empty linked lists.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Declare another constant in .assign() called linkedList with the value at arrayIndex in the hash map array.

This new constant linkedList will reference the linked list we want to add a value to.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

If the linked list at arrayIndex is empty, we should add the key-value pair to the list. The key-value pair should be stored in a node and become the head of the linked list before exiting the code.

A key-value pair can be stored in a node’s data property as an object with key and value properties:

{key: 'deer', value: 'forest animal'}

Data can be stored when the node is created by passing data to the Node constructor or it can be assigned to the data property later with dot notation and chaining.

// Using the Node constructor  
const first = new Node({'someKey', 'someValue'});  
   
// Using dot notation and chaining   
const second = new Node();  
second.data.key = 'someKey';  
second.data.value = 'someValue';

Add an if statement that checks if the linked list head exists. If it doesn’t exist, add the key-value pair as the head of the list.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, if the key-value pair has been added as the head node of the linked list we can exit the code. Inside the if statement use the return keyword to exit out of the code.

(This is important because we will using a loop to finish .assign() and we want to avoid an infinite loop.)

Checkpoint 5 Passed

answer: Collision.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null)

      .map(() => new LinkedList());

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    const linkedList = this.hashmap[arrayIndex];

    if (linkedList.head === null) {

      linkedList.addToHead({ key, value });

      return;

    }

  }

}

module.exports = HashMap;

HashMap.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null)

      .map(() => new LinkedList());

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    const linkedList = this.hashmap[arrayIndex];

    if (linkedList.head === null) {

      linkedList.addToHead({ key, value });

      return;

    }

  }

}

module.exports = HashMap;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Collisions: Looping**

We’ve added code to .assign() that takes care of an empty list, but what happens when there is a collision and there are already values stored at a particular index?

If there are already values stored in nodes at an index, we need to loop over each node in the list in order to determine how to proceed.

The two possibilities we’ll encounter while looping are:

* The key we are looking for and the key of the current node is the same, so we should overwrite the value
* No node in the linked list matches the key, so we should add the key-value pair to the list as the tail node

After both cases, if we haven’t already exited the loop, we should reset the loop’s condition.

With this in mind, the .assign() code for looping should look like this:

store the head node of the linked list in a variable current

while there is a current node

if the current node's key is the same as the key

store the key and value in current

if the current node is the tail node

store the key-value pair in the node after current

exit the loop

set current to the next node

### Instructions

**1.**

After the if statement in .assign() that checks for a head node, declare a variable that can be changed, current. Store the head node of the linked list in current.

We’ll use current to begin iterating over the linked list until we find the tail node.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

After declaring current in .assign(), iterate over the linked list to find the tail using a while loop.

While we haven’t visited all the nodes in a linked list, we should keep looping. Set the while loop condition to continue while current isn’t null.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Key-value pairs are stored in the data property of nodes. (To review how nodes are implemented, see the code for the Node class in **Node.js**.)

There are two possibilities when iterating over nodes:

* The current key and the node’s key are the same, and we should overwrite the node’s value with the current value
* The current key and the node’s key aren’t the same and we should check if there are more nodes in the linked list

Add an if statement in the while loop that does a strict equality check of the current node’s .key and key. If the two keys are the same, overwrite the current node’s key and value properties with the key-value pair we want to store.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Each node knows the node after it. We can use this to determine the tail of the linked list.

Outside of the last if statement, write a condition that checks if the current node is the tail node. Check what Node methods are available to you in the **Node.js** file.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

If the current node is the end of the linked list, there are no more nodes to loop over and check for a matching key.

Inside the if condition that checks for a tail node, set the next node after current to a new node with the key-value pair stored in it. (Check the **Node.js** file to see what methods are available to you to do this.) Then break out of the while loop.

Checkpoint 6 Passed

Stuck? Get a hint

**6.**

If we don’t reach the end of the linked list on this iteration, we need to check the next node.

Outside of the last if statement in the while loop set current to the next node in the linked list to continue the loop. Use the Node method that gets the next node in a linked list.

Checkpoint 7 Passed

Answer: HashMap.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null)

      .map(() => new LinkedList());

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    const linkedList = this.hashmap[arrayIndex];

    if (linkedList.head === null) {

      linkedList.addToHead({ key, value });

      return;

    }

    let current = linkedList.head;

    while (current) {

      if (current.data.key === key) {

        current.data = { key, value };

      }

      if (!current.getNextNode()) {

        const newNode = new Node({ key, value });

        current.setNextNode(newNode);

        break;

      }

      current = current.getNextNode();

    }

  }

}

module.exports = HashMap;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Collisions: Retrieving**

When we retrieve hash map values we also need to be aware that different keys could point to the same array index leading us to retrieve the wrong value.

To avoid this, we’ll search through the linked list at an index until we find a node with a matching key. If we find the node with the correct key, we’ll return the value otherwise we’ll return null.

The .retrieve() method will follow this logic:

store the hashed key in the constant arrayIndex

store the head node of a list in the variable current

while there is a valid node

if the current node's key matches the key

return the current node's value

set current to the next node in the list

return null

### Instructions

**1.**

In .retrieve(), declare a variable whose value can be changed, current. Assign it the head node of the linked list at arrayIndex to current.

We will search for the value we want to retrieve by checking each node in the linked list, starting at the head node.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Create a while loop that we’ll use to iterate over each node in the linked list until we either find the value we’re looking for or reach the end of the list.

Set the while condition to be the current node of the linked list.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Inside of the while loop add an if statement that checks if the key we received as an argument and the key of the current node are the same.

If both keys are the same, this means we’ve found the node with the correct value. Return the value stored in the current node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If the keys don’t match, we need to check the next node in the linked list. Outside of the if statement, set current to the next node in the linked list.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

If we’ve looped through the entire linked list without finding the value we want to retrieve, it means the value is not stored in the hash map. After the while loop, return null.

Checkpoint

Answer: HasHMap.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null)

      .map(() => new LinkedList());

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    const linkedList = this.hashmap[arrayIndex];

    if (linkedList.head === null) {

      linkedList.addToHead({ key, value });

      return;

    }

    let current = linkedList.head;

    while (current) {

      if (current.data.key === key) {

        current.data = { key, value };

      }

      if (!current.next) {

        current.next = new Node({ key, value });

        break;

      }

      current = current.next;

    }

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    let current = this.hashmap[arrayIndex].head;

    while (current) {

      if (current.data.key === key) {

        return current.data.value;

      }

      current = current.next;

    }

    return null;

  }

}

module.exports = HashMap;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Review**

**1.**

Create a constant that stores a hash map, birdCensus. We’ll use the hash map data structure to store all bird sightings. Give it an array size of 16.

Checkpoint 2 Passed

**2.**

It’s essential for our census that we know the type of bird seen and the location where it was spotted.

Assign the following key-value pairs to birdCensus:

* Key: 'mandarin duck', Value: 'Central Park Pond'
* Key: 'monk parakeet', Value: 'Brooklyn College'
* Key: 'horned owl', Value: 'Pelham Bay Park'

Checkpoint 3 Passed

**3.**

Retrieve the location for each of the three birds counted in birdCensus. Log them to the terminal.

Checkpoint 4 Passed

**4.**

Congratulations, you implemented a fully-functional hash map! Some things to consider:

* How would you delete a key-value pair from this hash map?
* Are there any other ways of handling collisions besides separate chaining? What would be the advantages or disadvantages of a method of avoiding separate chaining?

Answer: census.js

const HashMap = require('./HashMap');

const birdCensus = new HashMap(16);

birdCensus.assign('mandarin duck','Central Park Pond');

birdCensus.assign('monk parakeet', 'Brooklyn College');

birdCensus.assign('horned owl', 'Pelham Bay Park');

console.log(birdCensus.retrieve('mandarin duck'));

console.log(birdCensus.retrieve('monk parakeet'));

console.log(birdCensus.retrieve('horned owl'));

haspmap.js

const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {

  constructor(size = 0) {

    this.hashmap = new Array(size)

      .fill(null)

      .map(() => new LinkedList());

  }

  hash(key) {

    let hashCode = 0;

    for (let i = 0; i < key.length; i++) {

      hashCode += hashCode + key.charCodeAt(i);

    }

    return hashCode % this.hashmap.length;

  }

  assign(key, value) {

    const arrayIndex = this.hash(key);

    const linkedList = this.hashmap[arrayIndex];

    console.log(`Storing ${value} at index ${arrayIndex}`);

    if (linkedList.head === null) {

      linkedList.addToHead({ key, value });

      return;

    }

    let current = linkedList.head;

    while (current) {

      if (current.data.key === key) {

        current.data = { key, value };

      }

      if (!current.next) {

        current.next = new Node({ key, value });

        break;

      }

      current = current.next;

    }

  }

  retrieve(key) {

    const arrayIndex = this.hash(key);

    let current = this.hashmap[arrayIndex].head;

    while (current) {

      if (current.data.key === key) {

        console.log(`\nRetrieving ${current.data.value} from index ${arrayIndex}`);

        return current.data.value;

      }

      current = current.next;

    }

    return null;

  }

}

module.exports = HashMap;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output += `<tail>`;

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

TREE

Tree data structures are built using tree nodes (a variation on the nodes you created earlier) and are another way of storing information. Specifically, trees are used for data that has a hierarchical structure, such as a family tree or a computer’s file system. The tree data structure you are going to create is an excellent foundation for further variations on trees, including AVL trees, red-black trees, and binary trees.

**TREES: CONCEPTUAL**

**Trees Introduction**

Trees are an essential data structure for storing hierarchical data with a directed flow.

Similar to linked lists and graphs, trees are composed of nodes which hold data. The diagram represents nodes as rectangles and data as text.

Nodes also store references to zero or more **other tree nodes**. Data moves **down** from node to node. We depict those references as lines drawn between rectangles.

Trees are often displayed with a single node at the top and connected nodes branching downwards.

**Tree Detail**

Trees grow downwards in computer science, and a *root* node is at the very top. The root of this tree is /photos.

/photos references to two other nodes: /safari and /wedding. /safari and /wedding are *children* or *child* nodes of /photos.

Conversely, /photos is a *parent* node because it **has child nodes**.

/safari and /wedding share the same parent node, which makes them *siblings*.

Note that the /safari node is child (to /photos) **and** parent (to lion.jpg and giraffe.jpg). It’s extremely common to have nodes act as both parent and child to different nodes within a tree.

When a node has no children, we refer to it as a *leaf* node.

**Tree Varietals**

Trees come in various shapes and sizes depending on the dataset modeled.

Some are wide, with parent nodes referencing many child nodes.

Some are deep, with many parent-child relationships.

Trees can be both wide and deep, but each node will only ever have **at most** one parent; otherwise, they wouldn’t be trees!

Each time we move from a parent to a child, we’re moving down a *level*. Depending on the orientation we refer to this as the *depth* (counting levels down from the root node) or *height* (counting levels up from a leaf node).

**Binary Search Tree**

Constraints are placed on the data or node arrangement of a tree to solve difficult problems like efficient search.

A *binary tree* is a type of tree where each parent can have **no more than two children**, known as the *left child* and *right child*.

Further constraints make a *binary search tree*:

* Left child values must be lesser than their parent.
* Right child values must be greater than their parent.

The constraints of a binary search tree allow us to search the tree efficiently. At each node, we can discard **half** of the remaining possible values!

Let’s walk through locating the value 31.

1. Start at the root: 39
2. 31 < 39, we move to the left child: 23
3. 23 < 31, we move to the right child: 35
4. 31 < 35, we move to the left child: 31
5. We found the value 31!

In a dataset of **fifteen** elements, we only made **three** comparisons. What a deal!

**Tree Review**

Trees are useful for modeling data that has a hierarchical relationship which moves in the direction from parent to child. No child node will have more than one parent.

To recap some terms:

* root: A node which has no parent. One per tree.
* parent: A node which references other nodes.
* child: Nodes referenced by other nodes.
* sibling: Nodes which have the same parent.
* leaf: Nodes which have no children.
* level: The height or depth of the tree. Root nodes are at level 1, their children are at level 2, and so on.

**TREES: JAVASCRIPT**

**Introduction**

Trees are wonderful data structures that can model real life hierarchical information, including organizational charts, genealogical trees, computer file systems, HTML elements on a web page (also known as the Document Object Model, or DOM), state diagrams, and more.

A tree is composed of tree nodes. A tree node is a very simple data structure that contains:

* Data
* A list of children, where each child is itself a tree node

We can add data to and remove data from a tree and traverse it in two different ways:

* Depth-first, or
* Breadth-first

In this lesson, we’re going to implement the tree node data structure as a class in JavaScript.

### Instructions

**1.**

In **TreeNode.js**, you will find an empty TreeNode class. We will maintain the children of TreeNode as a JavaScript array. This will make it easier to add and remove a child.

Define a constructor that takes one parameter, data. Inside the constructor:

* define a data class property and assign it to the parameter, data
* define a children class property and assign it to an empty array.

Checkpoint 2 Passed

**2.**

Open **script.js**, instantiate a TreeNode class with argument of 1 and assign it to a const variable tree.

Display the contents of tree with console.log.

Checkpoint 3 Passed

ANSWER:TreeNode.JS

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

// instantiate your TreeNode class here

const tree = new TreeNode(1);

// display your TreeNode class here

console.log(tree);

**Adding a Child**

The next task is to add a child to our tree. Each child in our children array has to be an instance of a TreeNode, however we want to allow our user interface to accept adding data in other forms as well.

For instance, if our method to add a child is .addChild(), we want to accommodate calling tree.addChild(3) as well as tree.addChild(new TreeNode(3)).

### Instructions

**1.**

Below the constructor, define another method, .addChild() which takes one parameter, child.

Checkpoint 2 Passed

**2.**

Inside .addChild(), check if child is an instance of TreeNode. If it is, add child to the end of the children array. Otherwise, create a TreeNode instance for it before adding it to the children array.

Checkpoint 3 Passed

**3.**

Open **script.js**, and do the following:

* Add a child of value 15 to the tree object.
* Display the output of tree in the terminal

Checkpoint 4 Passed

**4.**

In **script.js**, do the following:

* Add another child by creating a TreeNode for it with value 30
* Add this child to the tree object.
* Display the output of tree on the terminal

Checkpoint 5 Passed

ANSWER: TreeNode,JS

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

const tree = new TreeNode(1);

console.log(tree);

tree.addChild(15);

console.log(tree);

tree.addChild(new TreeNode(30));

console.log(tree);

**Removing a Child**

Like with .addChild(), we want to provide a flexible interface for removing a child from a tree based on either its data or a TreeNode match. For example, if our method to remove a child is .removeChild(), we want to be able to execute the following:

const blue = 'blue';  
const green = new TreeNode('green');  
tree.addChild(blue);         // add data  
tree.addChild(green);        // add TreeNode  
tree.removeChild('blue');    // remove by data  
tree.removeChild(green);    // remove by TreeNode

The generic steps to execute in removing a child from a tree are as follows:

If target child is an instance of TreeNode,

Compare target child with each child in the children array

Update the children array if target child is found

Else

Compare target child with each child's data in the children array

Update the children array if target child is found

If target child is not found in the children array

Recursively call .removeChild() for each grandchild.

Because we implemented the children as an array, we can use the array .filter() method to update children. Like with .addChild(), we can also use instanceof to check if our target child is an instance of a TreeNode.

### Instructions

**1.**

Define a new method, .removeChild(), that takes one parameter, childToRemove.

**2.**

Inside .removeChild(), we want to remove the target child from the children array. Use the JavaScript .filter() method to filter out the elements that do not match the target child and reassign the array returned by .filter() back to the children array.

Do the following:

* Call the .filter() method on the children array and supply a callback function with a single argument, child, that does the following:
* If childToRemove is a TreeNode, return true if childToRemove is not equal to child, else return false.
* If childToRemove is not a TreeNode, return true if childToRemove is not equal to child‘s data, else return false.
* Reassign the return value of .filter() back to children.

**3.**

If the target child is not found in the children array, then we would have to descend another level by traversing each child in the array and repeat the process. How do we know that the target child has been removed from the children array? One way is to compare the length of the original children array with the updated children array that has been filtered.

Define a const variable called length and assign it to the length of the children array at the beginning of .removeChild() before the filtering.

**4.**

Compare length with the updated children‘s length after filtering. If they are the same, recursively call .removeChild() for each child in the children array.

**5.**

Now that we have completed implementing .removeChild(), let’s test it. Open **script.js**. A sample tree has been created for you and two children added, one by data and the other by TreeNode.

Do the following:

* Display the output of the tree.
* Remove the element in the tree by data and display the tree.
* Remove the element in the tree by TreeNode and display the tree.

ANSWER:

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

  removeChild(childToRemove) {

    const length = this.children.length;

    this.children = this.children.filter(child => {

      if (childToRemove instanceof TreeNode) {

        return childToRemove !== child;

      } else {

        return child.data !== childToRemove;

      }

    });

    if (length === this.children.length) {

      this.children.forEach(child => child.removeChild(childToRemove));

    }

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

const tree = new TreeNode(1);

tree.addChild(15);

const node = new TreeNode(30);

tree.addChild(node);

console.log(tree);

tree.removeChild(15);

console.log(tree);

tree.removeChild(node);

console.log(tree);

**Pretty Print**

Wouldn’t it be nice to be able to display the structure of our tree in a captivating visual way? We have provided a helpful method, .print() that will give you a formatted text display of our tree.

For example, a tree with 3 levels starting with root node 15, children 3, 12, 0, and grandchildren 6, 9, 19, 8, 10, 19 is displayed below:

15

-- 3

-- -- 6

-- -- 9

-- 12

-- -- 19

-- -- 8

-- 0

-- -- 10

-- -- 19

This method takes one parameter, level, which is initialized to 0, to enable printing the entire tree structure from the top to the bottom.

### Instructions

**1.**

Open **script.js**, and study how we add data in a sample tree. Then call .print() on the sample tree to see the output on the terminal.

ANSWER:

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

  removeChild(childToRemove) {

    const length = this.children.length;

    this.children = this.children.filter(child => {

      return childToRemove instanceof TreeNode

      ? child !== childToRemove

      : child.data !== childToRemove;

    });

    if (length === this.children.length) {

      this.children.forEach(child => child.removeChild(childToRemove));

    }

  }

  print(level = 0) {

    let result = '';

    for (let i = 0; i < level; i++) {

      result += '-- ';

    }

    console.log(`${result}${this.data}`);

    this.children.forEach(child => child.print(level + 1));

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

const tree = new TreeNode(1);

const randomize = () => Math.floor(Math.random() \* 20);

// add first-level children

for (let i = 0; i < 3; i++) {

  tree.addChild(randomize());

}

// add second-level children

for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 2; j++) {

    tree.children[i].addChild(randomize());

  }

}

// add third-level children

for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 2; j++) {

    for (let k = 0; k < 2; k++) {

      tree.children[i].children[j].addChild(randomize());

    }

  }

}

// pretty-print the tree

tree.print();

**Depth-first Tree Traversal**

Now that we can add nodes to our tree, the next step is to be able to traverse the tree and display its content. We can do this in one of two ways: depth-first or breadth-first.

Depth-first traversal visits the first child in the children array and that node’s children recursively before visiting its siblings and their children recursively. The algorithm is as follows:

For each node

Display its data

For each child in children, call itself recursively

Based on this tree displayed using .print():

15

-- 3

-- -- 6

-- -- 9

-- 12

-- -- 19

-- -- 8

-- 0

-- -- 10

-- -- 19

we can traverse it depth-wise to produce this result:

15

3

6

9

12

19

8

0

10

19

### Instructions

**1.**

In **TreeNode.js**, define a method, .depthFirstTraversal() below .print() that takes no parameters.

**2.**

Inside .depthFirstTraversal(), display the data of the current node with console.log.

**3.**

For each child in the children array, call .depthFirstTraversal() recursively.

**4.**

Open **script.js**. Do the following:

* Display the sample tree provided using the .print() method.
* Then, traverse the sample tree using the traversal method you have just created.
* Run the script.
* Study the results by comparing the output from .print() and .depthFirstTraversal(). Did you notice anything particular about the ordering of data from both methods?

ANSWER:

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

  removeChild(childToRemove) {

    const length = this.children.length;

    this.children = this.children.filter(child => {

      return childToRemove instanceof TreeNode

      ? child !== childToRemove

      : child.data !== childToRemove;

    });

    if (length === this.children.length) {

      this.children.forEach(child => child.removeChild(childToRemove));

    }

  }

  print(level = 0) {

    let result = '';

    for (let i = 0; i < level; i++) {

      result += '-- ';

    }

    console.log(`${result}${this.data}`);

    this.children.forEach(child => child.print(level + 1));

  }

  depthFirstTraversal() {

    console.log(this.data);

    this.children.forEach(child => child.depthFirstTraversal());

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

const tree = new TreeNode(15);

const randomize = () => Math.floor(Math.random() \* 20);

// add first-level children

for (let i = 0; i < 3; i++) {

  tree.addChild(randomize());

}

// add second-level children

for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 2; j++) {

    tree.children[i].addChild(randomize());

  }

}

tree.print();

tree.depthFirstTraversal();

**Breadth-first Tree Traversal**

Breadth-first traversal visits each child in the children array starting from the first child before visiting their children and further layers until the bottom level is visited. The algorithm is as follows:

Assign an array to contain the current root node

While the array is not empty

Extract the first tree node from the array

Display tree node's data

Append tree node's children to the array

Based on this tree displayed using .print():

15

-- 3

-- -- 6

-- -- 9

-- 12

-- -- 19

-- -- 8

-- 0

-- -- 10

-- -- 19

we can traverse it breadth-wise to produce this result:

15

3

12

0

6

9

19

8

10

19

Let’s implement our breadth-first traversal for TreeNode.

### Instructions

**1.**

Create a new method, .breadthFirstTraversal(), below .depthFirstTraversal() which takes no parameters.

**2.**

Inside .breadthFirstTraversal(), declare a let variable, queue and assign it to an array that contains the current node as its only element.

**3.**

Create a while loop evaluating if queue is not empty.

**4.**

Inside the while loop, extract the first element inside queue and assign it to a const variable, current. We do this so that we can display its data afterwards.

Log the data that belongs to current.

**5.**

While still inside the while loop, merge the current tree node’s children to the queue and reassign the merger to queue. We do this so that we can traverse the current node’s children after we finish traversing its siblings.

**6.**

Open **script.js**. Do the following:

* Display the sample tree provided using the pretty print method.
* Then, traverse the sample tree using the traversal method you have just created.
* Run the script.
* Study the results by comparing the output from .print() and .breadthFirstTraversal().

ANSWER:

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

  removeChild(childToRemove) {

    const length = this.children.length;

    this.children = this.children.filter(child => {

      return childToRemove instanceof TreeNode

      ? child !== childToRemove

      : child.data !== childToRemove;

    });

    if (length === this.children.length) {

      this.children.forEach(child => child.removeChild(childToRemove));

    }

  }

  print(level = 0) {

    let result = '';

    for (let i = 0; i < level; i++) {

      result += '-- ';

    }

    console.log(`${result}${this.data}`);

    this.children.forEach(child => child.print(level + 1));

  }

  depthFirstTraversal() {

    console.log(this.data);

    this.children.forEach(child => child.depthFirstTraversal());

  }

  breadthFirstTraversal() {

    let queue = [ this ];

    while (queue.length > 0) {

      const current = queue.shift();

      console.log(current.data);

      queue = queue.concat(current.children);

    }

  }

};

module.exports = TreeNode;

SCRIPT.JS

const TreeNode = require('./TreeNode');

const tree = new TreeNode(15);

const randomize = () => Math.floor(Math.random() \* 20);

// add first-level children

for (let i = 0; i < 3; i++) {

  tree.addChild(randomize());

}

// add second-level children

for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 2; j++) {

    tree.children[i].addChild(randomize());

  }

}

tree.print();

tree.breadthFirstTraversal();

* a TreeNode class that contains data and maintains a collection of TreeNode classes called children.
* an .addChild() method that adds a child to the tree as either data or TreeNode
* a .removeChild() method that removes a child from the tree as either data or TreeNode
* a .depthFirstTraversal() recursive method that fully traverses the tree with a top-down approach for each child of the tree
* a .breadthFirstTraversal() iterative method that fully traverses the tree a level at a time, instead of a child at a time

Congratulations!!

### Instructions

**1.**

In this exercise, we’ve constructed a sample menu tree, however some of the meal items are in the wrong category. Can you spot which ones and place them in the correct locations?

Open **script.js** and run it. You will see a pretty printout of the menu tree.

Menu

-- Breakfast

-- -- Cereal

-- -- BBQ Chicken

-- -- Oatmeal

-- Lunch

-- -- Soup

-- -- Sandwich

-- -- Lasagna

-- Dinner

-- -- Yogurt

-- -- Filet Mignon

-- -- Fish Florentine

**2.**

Two entries in the menu tree are dislocated. Write code to move each one to the correct location. Print the tree under the title Corrected Menu.

**3.**

Choose a tree traversal method whose output resembles the ordering of .print() and call it.

Answer:

const TreeNode = require('./TreeNode');

const menu = new TreeNode('Menu');

const entries = {

  'Breakfast' : [ 'Cereal', 'BBQ Chicken', 'Oatmeal' ],

  'Lunch' : [ 'Soup', 'Sandwich', 'Lasagna' ],

  'Dinner' : [ 'Yogurt', 'Filet Mignon', 'Fish Florentine' ]

};

const meals = Object.keys(entries);

for (let meal=0; meal < meals.length; meal++){

  menu.addChild(meals[meal]);

  const entrylist = entries[meals[meal]];

  entrylist.forEach( entry => {

    menu.children[meal].addChild(entry);

  });

}

menu.print();

// remove BBQ Chicken from Breakfast

menu.removeChild('BBQ Chicken');

// add BBQ Chicken to Dinner

menu.children[2].addChild('BBQ Chicken');

// remove Yogurt from Dinner

menu.removeChild('Yogurt');

// add Yogurt to Breakfast

menu.children[0].addChild('Yogurt');

console.log('------- Corrected Menu');

menu.print();

menu.depthFirstTraversal();

script.js

class TreeNode {

  constructor(data) {

    this.data = data;

    this.children = [];

  }

  addChild(child) {

    if (child instanceof TreeNode) {

      this.children.push(child);

    } else {

      this.children.push(new TreeNode(child));

    }

  }

  removeChild(childToRemove) {

    const length = this.children.length;

    this.children = this.children.filter(child => {

      return childToRemove instanceof TreeNode

      ? child !== childToRemove

      : child.data !== childToRemove;

    });

    if (length === this.children.length) {

      this.children.forEach(child => child.removeChild(childToRemove));

    }

  }

  print(level = 0) {

    let result = '';

    for (let i = 0; i < level; i++) {

      result += '-- ';

    }

    console.log(`${result}${this.data}`);

    this.children.forEach(child => child.print(level + 1));

  }

  depthFirstTraversal() {

    console.log(this.data);

    this.children.forEach(child => child.depthFirstTraversal());

  }

  breadthFirstTraversal() {

    let queue = [ this ];

    while (queue.length > 0) {

      const current = queue.shift();

      console.log(current.data);

      queue = queue.concat(current.children);

    }

  }

};

module.exports = TreeNode;

# Heaps

Heaps are another variation of the tree data structure and are adept at keeping track of the maximum or minimum value held within, referred to as max-heaps and min-heaps, respectively. Specifically, heaps are a type of binary tree, since each child node is either greater or less than its parent (depending on if it’s a max-heap or min-heap). They are efficient for accessing the root value, which will either be the max or min (again, depending on the type of heap) and inserting new values.

**Introduction to Heaps**

Heaps are used to maintain a maximum or minimum value in a dataset. Our examples use numbers since this is a straight-forward value, but heaps have many practical applications.

Imagine you have a demanding boss (hopefully this is theoretical!). They always want **the most important** thing done. Of course, once you finish the most important task, another one takes its place.

You can manage this problem using a **priority queue** to ensure you’re always working on the most pressing assignment and heaps are commonly used to create a priority queue.

Heaps tracking the maximum or minimum value are *max-heaps* or *min-heaps*. We will focus on min-heaps, but the concepts for a max-heap are nearly identical.

Think of the min-heap as a binary tree with two qualities:

* The root is the **minimum value** of the dataset.
* Every child’s value is **greater than or equal to its parent**.

These two properties are the defining characteristics of the min-heap. By maintaining these two properties, we can efficiently retrieve and update the minimum value.

**Heap Representations**

We can picture min-heaps as binary trees, where each node has **at most** two children. As we add elements to the heap, they’re added from left to right until we’ve filled the entire level.

At the top, we’ve filled the level containing 12 and 20. The next addition comes as the left child of 12, starting a new level in the tree. We would continue filling this level from left to right until 20 had its right child filled.

Conceptually, the tree representation is beneficial for understanding. Practically, we implement heaps in a sequential data structure like an array or list for efficiency.

Notice how by filling the tree from left to right; we’re leaving no gaps in the array. The location of each child or parent derives from a formula using the index.

* left child: (index \* 2) + 1
* right child: (index \* 2) + 2
* parent: (index - 1) / 2 — **not used on the root!**

**Adding an Element: Heapify Up**

* Sometimes you will add an element to the heap that violates the heap’s essential properties.
* We’re adding 3 as a left child of 11, which violates the min-heap property that children must be larger or equal to their parent.
* We need to restore the fundamental heap properties. This restoration is known as *heapify* or *heapifying*. We’re adding an element to the bottom of the tree and moving upwards, so we’re *heapifying up*.
* As long as we’ve violated the heap properties, we’ll swap the offending child with its parent until we restore the properties, or until there’s no parent left. If there is no parent left, that element becomes the new root of the tree.
* 3 swaps with 11, but there’s still work to do because now 3 is a child of 5. One more swap and we’ve restored the heap properties. The parent value, 2, is lesser than the child, 3. We can see that 3‘s children 5 and 14 are also greater.

**Removing an Element: Heapify Down**

Maintaining a minimum value is no good if we can never retrieve it, so let’s explore how to remove the root node.

In the diagram, you can see removing the top node itself would be messy: there would be two children orphaned! Instead, we’ll swap the root node, 2, with the bottom rightmost child: 20. The bottom rightmost child is simple to remove because it has no children.

Unfortunately, we’ve violated the heap property. 20 is now the root node, and that’s not the minimum value in the heap. We’ll *heapify down* to restore the heap property.

This process is similar to heapifying up, except we have two options (5 and 10) where we can make a swap. We’ll choose the **lesser of the two values** and swap 20 with 5. This is necessary for the heap property, if we had chosen to swap 20 with 10, then the minimum value would **not** be at the root. With 5 at the root, the root node is the minimum value in the heap again.

Another swap is required because 20 is greater than its children, so we swap 20 with 11.

Now 20 no longer has any children (it is a child of 11), and all other nodes in the heap only have parents with smaller values.

Just like that, we’ve retrieved the minimum value, allocated a *new* minimum, and maintained the heap property!

**Introduction**

A heap data structure is a specialized tree data structure that satisfies the heap condition:

* In a max-heap, for any given element, its parent’s value is greater than or equal to its value.
* In a min-heap, for any given element, its parent’s value is less than or equal to its value.

A heap data structure is commonly implemented as a binary tree. In this lesson, we’re going to implement a min-heap in JavaScript. Min-heaps efficiently keep track of the minimum value in a dataset, even as we add and remove elements.

Heaps enable solutions for complex problems such as finding the shortest path (Dijkstra’s Algorithm) or efficiently sorting a dataset (heapsort).

They’re an essential tool for confidently navigating some of the difficult questions posed in a technical interview.

By understanding the operations of a heap, you will have made a valuable addition to your problem-solving toolkit.

### Instructions

**1.**

The code in **script.js** creates a min-heap one element at a time from a random collection of numbers. It then removes the minimum value from the min-heap one at a time as well.

Run the code a few times to see the effects of adding and removing items in the min-heap printed to the screen.

Move to the next exercise when you’re ready to dig in further!

Checkpoint 2 Passed

Answer: script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// helper function to return a random integer

const randomize = () => Math.floor(Math.random() \* 40);

// populate minHeap with random numbers

for (let i = 0; i < 6; i++) {

  const num = randomize();

  console.log(`.. Adding value ${num}`);

  minHeap.add(num);

  console.log('Content of min-heap', minHeap.heap);

}

// return the minimum value in the heap until heap is empty

console.log('\n');

for (let i = 0; i < 6; i++) {

  console.log(`.. Removing minimum value ${minHeap.popMin()}`);

  console.log('Content of min-heap', minHeap.heap);

}

Minheap.js

class MinHeap {

  constructor() {

    this.heap = [null];

    this.size = 0;

  }

  add(value) {

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

  }

  popMin() {

    if (this.size === 0) {

      return null

    }

    const min = this.heap[1];

    this.heap[1] = this.heap[this.size];

    this.size--;

    this.heap.pop();

    this.heapify();

    return min;

  }

  bubbleUp() {

    let current = this.size;

    while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {

      this.swap(current, getParent(current));

      current = getParent(current);

    }

  }

  heapify() {

    let current = 1;

    let leftChild = getLeft(current);

    let rightChild = getRight(current);

    // Check that there is something to swap (only need to check the left if both exist)

    while (this.canSwap(current, leftChild, rightChild)){

      // Only compare left & right if they both exist

      if (this.exists(leftChild) && this.exists(rightChild)) {

        // Make sure to swap with the smaller of the two children

        if (this.heap[leftChild] < this.heap[rightChild]) {

          this.swap(current, leftChild);

          current = leftChild;

        } else {

          this.swap(current, rightChild);

          current = rightChild;

        }

      } else {

        // If only one child exist, always swap with the left

        this.swap(current, leftChild);

        current = leftChild;

      }

      leftChild = getLeft(current);

      rightChild = getRight(current);

    }

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

  exists(index) {

    return index <= this.size;

  }

  canSwap(current, leftChild, rightChild) {

    // Check that one of the possible swap conditions exists

    return (

      this.exists(leftChild) && this.heap[current] > this.heap[leftChild]

      || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]

    );

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

**MinHeap Class**

Our MinHeap class will store two pieces of information:

* An array of elements within the heap.
* A count of the elements within the heap.

To make our lives easier, we’ll always keep one element at the beginning of the array with the value null. By doing this, we can simplify our coding by always referencing our minimum element at index 1 instead of 0 and our last element at index this.size instead of this.size - 1.

const minHeap = new MinHeap();  
console.log(minHeap.heap);  
// [ null ]  
console.log(minHeap.size);  
// 0

### Instructions

**1.**

Within **MinHeap.js**, define the MinHeap class constructor with no parameter.

Inside the constructor:

* define a heap property as an array containing null, and
* define a size property instantiated to 0.

Checkpoint 2 Passed

**2.**

In **script.js**, make an instance of MinHeap and assign it to the const variable minHeap.

Checkpoint 3 Passed

**3.**

In **script.js**, display the heap content of minHeap.

Checkpoint 4 Passed

Answer:

Minheap.js

class MinHeap {

 constructor() {

   this.heap = [null];

   this.size = 0;

 }

}

module.exports = MinHeap;

script.js

const MinHeap = require('./MinHeap');

/\* make an instance \*/

const minHeap = new MinHeap;

console.log(minHeap.heap);

**Bubble Up Part I**

Our MinHeap needs to satisfy two conditions:

* The element at index 1 is the minimum value in the entire list.
* Every child element in the list must be larger than its parent.

Let’s define an .add() method which will allow us to add elements into the .heap array. We will also define .bubbleUp() which will do the work of maintaining the heap conditions as we add additional elements.

### Instructions

**1.**

Inside **MinHeap.js**, define a MinHeap class method, .add(), below the constructor to add an element to its heap. Within .add():

* define a parameter, value
* add value to end of the array in this.heap

Optionally, display in .add():

* a message showing the value to be added,
* the current content of the heap before method returns

Checkpoint 2 Passed

**2.**

After we added an element to the heap, we want to increase its heap count. Increment the size property by one.

Checkpoint 3 Passed

**3.**

Within **script.js**, call the .add() method with a value of 42. Run the test code within **script.js** to see the element 42 added to the heap.

Checkpoint 4 Passed

**4.**

Define another MinHeap class method, .bubbleUp(), below .add() whose task is to preserve the heap properties after an element is added to the heap. For now, log a message 'Bubble Up' inside the method.

Checkpoint 5 Passed

**5.**

We want to call .bubbleUp() each time we add an element. Return to the .add() method, and make a call to .bubbleUp()

Checkpoint 6 Passed

**6.**

Rerun the test code in **script.js**. What do you see as output?

Checkpoint 7 Passed

Answer:minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  add(value) {

    this.heap.push(value);

    console.log(`.. Adding ${value}`);

    console.log(this.heap);

    this.size++;

    this.bubbleUp();

  }

  bubbleUp() {

    console.log('Bubble Up');

  }

}

module.exports = MinHeap;

script.js

const MinHeap = require('./MinHeap');

// instantiate MinHeap and assign to minHeap

const minHeap = new MinHeap();

// display content of minHeap

console.log('Content of heap', minHeap.heap);

minHeap.add(42);

**Parent and Child Elements**

Great work so far! Our MinHeap adds elements to the internal heap, keeps a running count, and has the beginnings of .bubbleUp().

Before we dive into the logic for .bubbleUp(), let’s review how heaps track elements. We use an array for storing internal elements, but we’re modeling it on a binary tree, where every parent element has up to two child elements.

Child and parent elements are determined by their relative indices within the internal heap. By doing some arithmetic on an element’s index, we can determine the indices for parent and child elements (if they exist).

* Parent: (index / 2), rounded down
* Left Child: index \* 2
* Right Child: (index \* 2) + 1

These calculations are important for the efficiency of the heap, but they’re not necessary to memorize, so we have provided three helper functions: getParent(), getLeft(), and getRight() in **MinHeap.js**.

These helpers take an index as the sole parameter and return the corresponding parent or child index.

console.log(myHeap.heap)   
// returns [null, 10, 13, 21, 61, 22, 23, 99]  
   
getParent(4); // returns (4 / 2) == 2  
   
getLeft(3);   // returns (3 \* 2) == 6  
   
getRight(3);  // returns (3 \* 2) + 1 == 7

### Instructions

**1.**

In **script.js**, test out the three helper functions above. A sample populated MinHeap has been provided for you. Replace null with the correct way to access the values of the parent, left child and right child indices.

Answer:script.js

const { MinHeap, getParent, getLeft, getRight } = require('./MinHeap');

// instantiate MinHeap and assign to minHeap

const minHeap = new MinHeap();

// sample content of minHeap

minHeap.heap = [ null, 10, 13, 21, 61, 22, 23, 99 ];

// display content of minHeap

console.log(minHeap.heap);

// display the current value, its parent value, left child value and right child value

// replace null with the correct way to access the values of the parent, left child and right child

const current = 3;

const currentValue = minHeap.heap[current];

console.log(`Current value of ${current} is ${currentValue}`);

console.log(`Parent value of ${currentValue} is ${minHeap.heap[getParent(current)]}`);

console.log(`Left child value of ${currentValue} is ${minHeap.heap[getLeft(current)]}`);

console.log(`Right child value of ${currentValue} is ${minHeap.heap[getRight(current)]}`);

minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  add(value) {

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

    console.log(this.heap);

  }

  bubbleUp() {

    console.log('Bubble Up');

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = {

  MinHeap,

  getParent,

  getLeft,

  getRight

};

**Bubble Up Part II**

Now that we understand how to determine the relationship of elements with the internal heap, we’re ready to finish .bubbleUp().

In a min-heap, we need to ensure that every child is greater in value than its parent. Let’s add an element to the following heap.

console.log(minHeap.heap)  
// returns [null, 10, 13, 21, 61, 22, 23, 99]  
   
heap.add(12)

( new\_element )

{ parent\_element }

[null, 10, 13, 21, {61}, 22, 23, 99, (12)]

Oh no! We’ve violated the min-heap condition because 12‘s parent, 61, is greater than its child, 12.

To fix this, we will exchange the parent and the child elements.

before

[null, 10, 13, 21, {61}, 22, 23, 99, (12)]

SWAP 12 and 61

after

[null, 10, 13, 21, (12), 22, 23, 99, {61}]

12‘s parent is now 13 and it violates the min-heap condition. To fix this, we continue moving upwards swapping parent-child values.

before

[null, 10, {13}, 21, (12), 22, 23, 99, 61]

SWAP 12 and 13

after

[null, 10, (12), 21, {13}, 22, 23, 99, 61]

12‘s parent is now 10 and no longer violates the min-heap condition. We’ve restored the heap!

[null, {10}, (12), 21, 13, 22, 23, 99, 61]

The child, 12, is greater than the parent, 10!

Let’s recap our strategy for .bubbleUp():

Set the current element index to be the last index of heap

While current element is valid and its value is less than its parent's value

Swap the indexes

Update the current element index to be its parent index

### Instructions

**1.**

.bubbleUp() is called by .add() which appends an element to the internal heap property. Hence, we need to keep track of the added element.

Inside .bubbleUp(), declare a let current variable that will point to the added element’s index. Initialize current to the added element’s current location, which is the end of heap.

Checkpoint 2 Passed

**2.**

In .bubbleUp(), set up a while loop that will run as long as it meets these 2 conditions:

* There is a valid current index. A valid current index is anything greater than 1.
* The value at the current index is less than its parent’s value. This will violate the min-heap condition and will trigger swapping values. Use a helper method to derive the parent index.

Checkpoint 3 Passed

**3.**

Inside the while loop, swap the parent index and the current index using the helper method, .swap() that has been provided for you. Pass both current and parent indices to .swap().

Optionally, display:

* the content of the current heap and
* a message that shows swapping will occur between the current index and its parent before the actual swap.

Checkpoint 4 Passed

**4.**

The last thing to do inside the while loop is to update the current index to be its parent’s index, since we are progressing upwards, or bubbling up, the binary tree model of the min-heap.

Checkpoint 5 Passed

**5.**

Open **script.js** and run the code to enjoy the fruits of your labor! You should find the smallest value at the beginning of the heap at index 1.

Checkpoint 6 Passed

Answer: minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  add(value) {

    console.log(`.. adding ${value}`);

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

    console.log(`added ${value} to heap`, this.heap);

  }

  bubbleUp() {

    let current = this.size;

    while (current > 1 && this.heap[current] < this.heap[getParent(current)]) {

      console.log('..', this.heap);

      console.log(`.. swap index ${current} with ${getParent(current)}`);

      this.swap(current, getParent(current));

      current = getParent(current);

    }

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// helper function to return a random integer

function randomize() { return Math.floor(Math.random() \* 40); }

// populate minHeap with random numbers

for (let i=0; i < 6; i++) {

  minHeap.add(randomize());

}

// display the bubbled up numbers in the heap

console.log('Bubbled Up', minHeap.heap);

**Removing the Min**

Min-heaps would be useless if we couldn’t retrieve the minimum value. We’ve gone through a lot of work to maintain that value because we’re going to need it!

Our goal is to efficiently remove the minimum element from the heap. You’ll recall that we always locate the minimum element at index 1 (a placeholder element occupies index 0).

Our internal heap mirrors a binary tree. There’s a delicate balance of parent and child relationships we would ruin by directly removing the minimum.

console.log(minHeap.heap)  
// [null, 10, 21, 13, 61, 22, 23, 99]  
minHeap.popMin()  
// 10  
// [null, ???, 21, 13, 61, 22, 23, 99]

We need to remove an element that has no children, in this case, the last element. If we swap the minimum with the last element, we can easily remove the minimum from the end of the list.

[None, (10), 21, 13, 61, 22, 23, {99}]

minHeap.popMin()

SWAP minimum with last

[None, {99}, 21, 13, 61, 22, 23, (10)]

remove minimum

[None, 99, 21, 13, 61, 22, 23]

10

Terrific! We removed the minimum element with minimal disruption. Unfortunately, our heap is out of shape again with 99 sitting where the minimum element should be. We will solve this in exercises to come…

### Instructions

**1.**

To retrieve the minimum value of our heap, we need to define a class method.

* Define .popMin() below the constructor within our MinHeap class.
* Within .popMin(), check if our heap is empty. If it is, return null.

Checkpoint 2 Passed

**2.**

Next, we want to:

* exchange the last element of the heap with the minimum element at index 1 using .swap()
* remove the last element from the heap, and save it in a const min variable
* decrement the heap size.

Checkpoint 3 Passed

**3.**

Display:

* a message to show that the first element ${this.heap[1]} is swapped with the last element, ${this.heap[this.size]} (do this before the actual swap)
* a message that shows that the minimum element has been removed followed by the content of the heap; use the stringRemove at the beginning of the message.

Checkpoint 4 Passed

**4.**

Lastly, return the min variable in .popMin().

Checkpoint 5 Passed

**5.**

Open **script.js** and run the test code.

Checkpoint

Answer: minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  popMin() {

    if (this.size === 0) {

      return null;

    }

 console.log(`\n.. Swap ${this.heap[1]} with last element ${this.heap[this.size]}`);

    this.swap(1, this.size);

    const min = this.heap.pop();

    this.size--;

    console.log(`.. Removed ${min} from heap`);

    console.log('..',this.heap);

    return min;

  }

  add(value) {

    console.log(`.. adding ${value}`);

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

    console.log(`added ${value} to heap`, this.heap);

  }

  bubbleUp() {

    let current = this.size;

    while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {

      console.log('..', this.heap);

      console.log(`.. swap index ${current} with ${getParent(current)}`);

      this.swap(current, getParent(current));

      current = getParent(current);

    }

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// helper function to return a random integer

function randomize() { return Math.floor(Math.random() \* 40); }

// populate minHeap with random numbers

for (let i=0; i < 6; i++) {

  minHeap.add(randomize());

}

// display the bubbled up numbers in the heap

console.log('Bubbled Up', minHeap.heap);

// remove the minimum value from heap

minHeap.popMin();

**Heapify I**

We’ve retrieved the minimum element but left our MinHeap in disarray. There’s no reason to get discouraged; we’ve handled this type of problem before, and we can get our MinHeap back in shape!

We’ll define a method, .heapify(), which performs a similar role to .bubbleUp(), except now we’re moving down the tree instead of up. The current element is a parent that can have either a left child or two children, but not just a right child.

We have written a helper method, .canSwap(), to return true if swapping can occur for either child and false otherwise:

canSwap(current, leftChild, rightChild) {  
  // Check that one of the possible swap conditions exists  
  return (this.exists(leftChild) && this.heap[current] > this.heap[leftChild]   
  || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]  
    );  
  }

To maintain the min-heap condition, the parent value has to be less than both its child values. To see if a swap is necessary, starting with the left child, we first check that the child exists and then whether the min-heap condition is broken, i.e. the current element has a value greater than that child’s value. If the left child does not break the min-heap condition, the same check is performed on the right child.

### Instructions

**1.**

Define an empty .heapify() method below .bubbleUp() in MinHeap.

Checkpoint 2 Passed

**2.**

We are going to heapify beginning from the index that always points to the minimum value. Declare a let current which points to index 1.

At this stage, index 1 is pointing to the out-of-place value we swapped in while removing the minimum.

Checkpoint 3 Passed

**3.**

We are going to use .canSwap() as we traverse each element in our heap tree. Since .canSwap() takes 3 arguments: current index, left child index, and right child index, we need to add two more local variables.

* Declare two local variables leftChild and rightChild and assign them to their appropriate values.
* Write a while loop that calls .canSwap()
* At the bottom of the while loop, update the leftChild and rightChild to their appropriate values so that the loop will not run infinitely.

In later exercises, we will continue filling the while loop to restore the heap.

Checkpoint 4 Passed

Answer: minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  popMin() {

    if (this.size === 0) {

      return null

    }

    console.log(`\n.. Swap ${this.heap[1]} with last element ${this.heap[this.size]}`);

    this.swap(1, this.size);

    const min = this.heap.pop();

    this.size--;

    console.log(`.. Removed ${min} from heap`);

    console.log('..',this.heap);

    return min;

  }

  add(value) {

    console.log(`.. adding ${value}`);

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

    console.log(`added ${value} to heap`, this.heap);

  }

  bubbleUp() {

    let current = this.size;

    while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {

      console.log(`.. swap ${this.heap[current]} with parent ${this.heap[getParent(current)]}`);

      this.swap(current, getParent(current));

      console.log('..', this.heap);

      current = getParent(current);

    }

  }

  heapify() {

    let current = 1;

    let leftChild = getLeft(current);

    let rightChild = getRight(current);

    while (this.canSwap(current, leftChild, rightChild)) {

      leftChild = getLeft(current);

      rightChild = getRight(current);

    }

  }

  exists(index) {

    return index <= this.size;

  }

  canSwap(current, leftChild, rightChild) {

    // Check that one of the possible swap conditions exists

    return (

      this.exists(leftChild) && this.heap[current] > this.heap[leftChild]

      || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]

    );

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// helper function to return a random integer

function randomize() { return Math.floor(Math.random() \* 40); }

// populate minHeap with random numbers

for (let i=0; i < 6; i++) {

  minHeap.add(randomize());

}

// display the bubbled up numbers in the heap

console.log('Bubbled Up', minHeap.heap);

// remove the minimum value from heap

minHeap.popMin();

**Heapify II**

In .bubbleUp(), we were always comparing our element with its parent. In .heapify(), we have potentially two options: the left child and the right child.

Which should we choose? We’ll use an example to think it through. Imagine we have a heap with four elements:

console.log(minHeap.heap)  
// [null, 21, 36, 58, 42]  
minHeap.popMin()  
// 21  
// [null, 42, 36, 58]  
// Should we swap 42 with 36 or 58?

We want to swap with the smaller of the two children, otherwise, we wouldn’t maintain our heap condition!

### Instructions

**1.**

In .heapify() at the beginning of the while loop, check to see if leftChild and rightChild both exist. Use the helper method .exists() to check for the existence of an element at a particular index.

Checkpoint 2 Passed

**2.**

If both children exist, we need to only swap with the smaller of the two. Inside the if statement that checks for the existence of both children, compare the left child’s value with the right child’s value.

If the left child is smaller than the right child:

* swap the current element with the left child
* update the current element index to be the left child

Otherwise, if the right child is smaller than the left child:

* swap the current element with the right child
* update the current element index to be the right child

Caveat: Executing “Run” may cause an infinite while loop if the if and else statements are left blank or have incorrect content. You can refresh the page to stop it.

Checkpoint 3 Passed

**3.**

If only one child exists, it has to be the left child. Write an else block to the outer if statement that:

* swaps the current element with the left child, and
* updates the current element index to be the left child

Caveat: Executing “Run” may cause an infinite while loop if the else statement is left blank or has incorrect content.

Checkpoint 4 Passed

**4.**

Go back into .popMin() and make a call to .heapify() before we return min.

Checkpoint 5 Passed

**5.**

Open **script.js** and run the code inside. Study the output to strengthen your understanding of the .heapify() method.

Checkpoint

Answer: minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  popMin() {

    if (this.size === 0) {

      return null

    }

    console.log(`\n.. Swap ${this.heap[1]} with last element ${this.heap[this.size]}`);

    this.swap(1, this.size);

    const min = this.heap.pop();

    this.size--;

    console.log(`.. Removed ${min} from heap`);

    console.log('..',this.heap);

    this.heapify();

    return min;

  }

  add(value) {

    console.log(`.. adding ${value}`);

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

    console.log(`added ${value} to heap`, this.heap);

  }

  bubbleUp() {

    let current = this.size;

    while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {

      console.log(`.. swap ${this.heap[current]} with parent ${this.heap[getParent(current)]}`);

      this.swap(current, getParent(current));

      console.log('..', this.heap);

      current = getParent(current);

    }

  }

  heapify() {

    let current = 1;

    let leftChild = getLeft(current);

    let rightChild = getRight(current);

    while (this.canSwap(current, leftChild, rightChild)) {

      while (this.canSwap(current, leftChild, rightChild)) {

      if (this.exists(leftChild) && this.exists(rightChild)) {

        if (this.heap[leftChild] < this.heap[rightChild]) {

          this.swap(current, leftChild);

          current = leftChild;

        } else {

          this.swap(current, rightChild);

          current = rightChild;

        }

      } else {

        this.swap(current, leftChild);

        current = leftChild;

      }

      leftChild = getLeft(current);

      rightChild = getRight(current);

    }

      leftChild = getLeft(current);

      rightChild = getRight(current);

    }

  }

  exists(index) {

    return index <= this.size;

  }

  canSwap(current, leftChild, rightChild) {

    // Check that one of the possible swap conditions exists

    return (

      this.exists(leftChild) && this.heap[current] > this.heap[leftChild]

      || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]

    );

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// helper function to return a random integer

function randomize() { return Math.floor(Math.random() \* 40); }

// populate minHeap with random numbers

for (let i=0; i < 6; i++) {

  minHeap.add(randomize());

}

// display the bubbled up numbers in the heap

console.log('Bubbled Up', minHeap.heap);

// remove the minimum value from heap

for (let i=0; i < 6; i++) {

  minHeap.popMin();

  console.log('Heapified', minHeap.heap);

}

You’ve implemented a min-heap in JavaScript, and that’s no small feat (although it could efficiently track the smallest feat).

To recap: MinHeap tracks the minimum element as the element at index 1 within an internal Javascript array.

When adding elements, we use .bubbleUp() to compare the new element with its parent, making swaps if it violates the heap condition: children must be greater than their parents.

When removing the minimum element, we swap it with the last element in the heap. Then we use .heapify() to compare the new root with its children, swapping with the smaller child if necessary.

Heaps are useful because they’re efficient in maintaining their heap condition. Building a heap using elements that decrease in value would ensure that we continually violate the heap condition. How many swaps would that cause?

### Instructions

**1.**

Run the code in **script.js** to see how many swaps are made in a dataset of 10000 elements! We added extra lines of code to keep a count on the number of swaps in .bubbleUp() and .heapify() and log a message when the heap size has reached the 10000th element in .bubbleUp() and 9999 elements in .heapify().

The number of swaps can be at most the height of the binary tree. The relationship between the maximum number of nodes, N, of a binary tree and the height, h, is:

N = 2h+1 - 1*N*=2*h*+1−1

For a height of 13, the maximum number of nodes is

214 - 1 = 16383.214−1=16383.

For a height of 12, the maximum number of nodes is

213 - 1 = 8191.213−1=8191.

Since 10000 falls between 8191 and 16383, the number of swaps can be at most 13.

Checkpoint

Answer : script.js

// import MinHeap class

const MinHeap = require('./MinHeap');

// instantiate a MinHeap class

const minHeap = new MinHeap();

// populate minHeap with descending numbers from 10001 to 1

console.log('Adding');

for (let i=10000; i >=1; i--) {

  minHeap.add(i);

}

// remove the minimum value from heap

console.log('Removing');

console.log('Minimum value = ' + minHeap.popMin());

minheap.js

class MinHeap {

  constructor() {

    this.heap = [ null ];

    this.size = 0;

  }

  popMin() {

    if (this.size === 0) {

      return null

    }

    const min = this.heap[1];

    this.heap[1] = this.heap[this.size];

    this.heap.pop();

    this.size--;

    this.heapify();

    return min;

  }

  add(value) {

    this.heap.push(value);

    this.size++;

    this.bubbleUp();

  }

  bubbleUp() {

    let current = this.size;

    let swapCount = 0;

    while (current > 1 && this.heap[getParent(current)] > this.heap[current]) {

      this.swap(current, getParent(current));

      current = getParent(current);

      swapCount++;

    }

    if (this.size == 10000) {

      console.log(`Heap of ${this.size} elements restored with ${swapCount} swaps`);

    }

  }

  heapify() {

    let current = 1;

    let leftChild = getLeft(current);

    let rightChild = getRight(current);

    let swapCount = 0;

    while (this.canSwap(current, leftChild, rightChild)) {

      // Only compare left & right if they both exist

      if (this.exists(leftChild) && this.exists(rightChild)) {

        // Make sure to swap with the smaller of the two children

        if (this.heap[leftChild] < this.heap[rightChild]) {

          this.swap(current, leftChild);

          current = leftChild;

    swapCount++;

        } else {

          this.swap(current, rightChild);

          current = rightChild;

    swapCount++;

        }

      } else {

        // If only one child exist, always swap with the left

        this.swap(current, leftChild);

        current = leftChild;

  swapCount++;

      }

      leftChild = getLeft(current);

      rightChild = getRight(current);

    }

    if (this.size == 9999) {

      console.log(`Heap of ${this.size} elements restored with ${swapCount} swaps`);

    }

  }

  exists(index) {

    return index <= this.size;

  }

  canSwap(current, leftChild, rightChild) {

    // Check that one of the possible swap conditions exists

    return (

      this.exists(leftChild) && this.heap[current] > this.heap[leftChild]

      || this.exists(rightChild) && this.heap[current] > this.heap[rightChild]

    );

  }

  swap(a, b) {

    [this.heap[a], this.heap[b]] = [this.heap[b], this.heap[a]];

  }

}

const getParent = current => Math.floor((current / 2));

const getLeft = current => current \* 2;

const getRight = current => current \* 2 + 1;

module.exports = MinHeap;

# Graphs

Graphs are used to represent data points, or vertices, that are connected by edges. Common applications for graphs are things like maps, where each location is a vertex, and each path, or road, between the locations is an edge. Graphs can be directed (a one-way street) or undirected (a two-way street), as well as weighted or unweighted (think of the length of each street as a potential measurement of weight).

**Introduction to Graphs**

Graphs are the perfect data structure for modeling networks, which make them an indispensable piece of your data structure toolkit. They’re composed of nodes, or *vertices*, which hold data, and *edges*, which are a connection between two vertices. A single node is a *vertex*.

Consider a map of the area where you live. As a graph, we could model bus stops as vertices, with bus routes between stops functioning as the edges.

What about the internet? Web pages can be vertices, and the hyperlinks which connect them are edges.

Real-world relationships modeled as graphs are numerous, making them an essential concept to master.

**To Connect, or Not to Connect?**

Graphs have varying degrees of connection. The higher the ratio of edges to vertices, the more connected the graph.

This graph represents a social network; people are vertices and edges are friendships. Ted is *adjacent* to Patty, Ron, and Alice because an edge **directly** connects them.

We use a single line for an edge, but these friendships are **bi-directional**. Patty is friends with Ron and Ron is friends with Patty.

A *path* is vertices which are connected by any number of intermediate edges. The paths from Alice to Patty could go Alice to Ted to Patty **or**, Alice to Ted to Ron to Patty.

No path exists between Sally and Ted. When no path exists between two vertices, a graph is *disconnected*.

**You're Going to Carry that Weight**

We’re building a graph of favorite neighborhood destinations (vertices) and routes (edges), but not all edges are equal. It takes longer to travel between Gym and Museum than it does to travel between Museum and Bakery.

This is a *weighted* graph, where edges have a number or cost associated with traveling between the vertices. When tallying the cost of a path, we add up the **total** cost of the edges used.

These costs are essential to algorithms that find the shortest distance between two vertices.

Gym and Library are adjacent, there’s one edge between them, but there’s less total cost to travel from Gym to Bakery to Library (10 vs. 9).

In a weighted graph, the shortest path is not always the least expensive.

**Directed Graphs**

Imagine you’re a superhero escaping a villain’s lair. As you move from perilous room to perilous room, the doors close immediately behind you, barring any return.

For this dramatic example, we need a *directed* graph, where edges restrict the direction of movement between vertices.

We can move from spikes to lasers, but not from lasers to spikes. This differs from earlier examples when every edge was bi-directional.

Note the path spikes to lasers to piranhas to spikes. This path is a *cycle*, because it ends on the vertex where it began: spikes.

**Representing Graphs**

We typically represent the vertex-edge relationship of a graph in two ways: an adjacency list or an adjacency matrix.

An adjacency matrix is a table. Across the top, every vertex in the graph appears as a column. Down the side, every vertex appears again as a row. Edges can be bi-directional, so each vertex is listed twice.

To find an edge between B and P, we would look for the B row and then trace across to the P column. The contents of this cell represent a possible edge.

Our diagram uses 1 to mark an edge, 0 for the absence of an edge. In a weighted graph, the cell contains the cost of that edge.

In an adjacency list, each vertex contains a list of the vertices where an edge exists. To find an edge, one looks through the list for the desired vertex.

**Reviewing Key Terms**

Graphs are an essential data structure in computer science for modeling networks. Let’s review some key terms:

* vertex: A node in a graph.
* edge: A connection between two vertices.
* adjacent: When an edge exists between vertices.
* path: A sequence of one or more edges between vertices.
* disconnected: Graph where at least two vertices have no path connecting them.
* weighted: Graph where edges have an associated cost.
* directed: Graph where travel between vertices can be restricted to a single direction.
* cycle: A path which begins and ends at the same vertex.
* adjacency matrix: Graph representation where vertices are both the rows and the columns. Each cell represents a possible edge.
* adjacency list: Graph representation where each vertex has a list of all the vertices it shares an edge with.

Exercise:

**Introduction to Graphs**

In this lesson, we’ll take an object-oriented approach to build an implementation of the graph data structure in JavaScript. With three classes, Edge, Vertex, and Graph, we can implement a variety of graphs that satisfy the requirements of many algorithms. Remember that a Graph consists of vertices and their corresponding edges.

For this lesson, we want our Graph class to be flexible enough to support directed, undirected, weighted, and unweighted graphs. We will provide you with an Edge class that connects two vertices, along with the weight of the connection (to support weighted graphs).

With this in mind, we will create our Graph with the following requirements:

* A Vertex can store any data.
* A Vertex maintains a list of connections to other vertices, represented by a list of Edge instances.
* A Vertex can add and remove edges going to another Vertex.
* A Graph stores all of its vertices, represented by a list of Vertex instances.
* A Graph knows if it is directed or undirected.
* A Graph knows if it is weighted or unweighted.
* A Graph can add and remove its own vertices.
* A Graph can add and remove edges between stored vertices.

Let’s start with familiarizing ourselves with the classes that we will build in **Vertex.js** and **Graph.js**. We already set up .print() methods for you that will print out the state of the graph structure. Don’t worry about the class in **Edge.js** yet. We will use it to connect the vertices in a later exercise.

To keep the concepts grounded in a real-world application, we’ll build a transportation network of railroads and train stations as we go.

### Instructions

**1.**

Let’s start by setting up the constructor for our Vertex class. When a vertex is first created, it should hold any given data, and it should have an empty list of edges because it does not have any connections.

In the constructor, expect a data parameter and set it to the data class property. Then, set the edges class property to an empty array.

Checkpoint 2 Passed

**2.**

Moving on to the constructor for our Graph class, a graph is essentially a collection of vertices and edges. Our graph only needs to keep track of a list of vertices.

In the Graph class in **Graph.js**, create a constructor that takes no parameters. Since a graph doesn’t have any vertices when it is first created, set the vertices property to an empty array in the constructor.

Checkpoint 3 Passed

Answer: vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data) || [];

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

graph.js

cconst Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor() {

    this.vertices = [];

  }

  print() {

    const vertexList = this.vertices || [];

    vertexList.forEach(vertex => vertex.print());

  }

}

module.exports = Graph;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Adding Vertices**

Now that we have set up our data structures, we can provide an easier way to manage the graph’s list of vertices. This gives us an opportunity to abstract out the places that use the Vertex class.

Currently, we would have to manually create a new Vertex instance and add it into the Graph’s list of vertices to populate the graph. If we create an .addVertex() method in the Graph class, it simplifies the process of adding a vertex to the graph. This alleviates the burden of knowing how the Vertex class should interact with the Graph class for whoever is using our Graph. They only need to interact with the Graph class.

### Instructions

**1.**

Inside the Graph class, add an .addVertex() method that expects a single parameter, data, which contains the data of vertex to create. Using this argument, create a Vertex instance and add it to the Graph’s list of vertices.

We will want to return the newly created Vertex to signal to the method caller that a vertex was successfully created and added to the list.

Checkpoint 2 Passed

**2.**

Great! Now we can set up our train network graph and see what it looks like with our train station vertices.

Underneath the Graph class, create a Graph instance and assign it to the trainNetwork variable. Then use the .addVertex() method to add two train stations with the names, 'Atlanta' and 'New York'. Assign the newly created vertices to the variables atlantaStation and newYorkStation, respectively. We will use these variables later on.

Call the .print() method on the trainNetwork. We should see our graph with two vertices inside it. They should be labeled Atlanta and New York, respectively.

Checkpoint 3 Passed

Answer: graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor() {

    this.vertices = [];

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph();

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

trainNetwork.print();

module.exports = Graph;

vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Removing Vertices**

We also want our Graph to manage its own vertex removal, just like how it handles its own vertex creation.

We will use the .removeVertex() method to look for the requested vertex and remove it from the list of vertices.

### Instructions

**1.**

Inside the Graph class, implement the .removeVertex() method that accepts the vertex to be removed as a parameter. Iterate through the list of vertices and remove the vertex that is strictly equal to the vertex given in the parameter.

Checkpoint 2 Passed

**2.**

Underneath our Graph class, let’s remove the Atlanta vertex we added in the previous exercise using the trainNetwork‘s .removeVertex() method. Remember to do it before the call to .print() so we can see what the resulting graph looks like.

We should see our graph with only the New York vertex remaining, and no edges.

Checkpoint 3 Passed

Answer: graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor() {

    this.vertices = [];

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  /\* 1 answer\*/

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph();

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

/\* 2nd answer\*/

trainNetwork.removeVertex(atlantaStation);

trainNetwork.print();

module.exports = Graph;

vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Connecting Vertices with Edges**

Since we can add vertices to our graph, we should be able to connect them together. We want to provide this functionality in the Graph class to add a layer of abstraction that will simplify adding edges, similar to how we abstracted vertex creation. This is where our Edge class in **Edge.js** will come in handy. Go ahead and take a look at the class.

The start and end properties mark the vertices that the edge connects. If the graph is directed, we can indicate the direction the edge points (towards the end vertex).

We will create an .addEdge() method in the Vertex class that connects the vertices together by creating an Edge and adding it to the vertices’ list of edges. When the Edge is created, it expects the two Vertex instances, which is how the Edge tracks the connection between the two vertices .

Then, we will use this method in the Graph‘s .addEdge() method to create edges going in both directions between the two given vertices. Even though this graph is undirected, we want to create two edges going in both directions so it is easier to traverse.

### Instructions

**1.**

In our Vertex class, create the .addEdge() method that expects a vertex parameter, which will represent the other end of the edge. It must be an instanceof a Vertex, otherwise we should throw an error. Then, create an Edge instance to represent the connection from this vertex to the ending vertex.

Add the Edge instance to the vertex’s list of edges to open up our first connection from one vertex to another.

Checkpoint 2 Passed

**2.**

We’re ready to connect vertices with edges through our Graph class. In the Graph class, create an .addEdge() method, which will create edges between the parameters, vertexOne and vertexTwo.

If the parameters are both an instanceof a Vertex, use the vertices’ .addEdge() method to create an edge between the other vertex. Remember to add edges between both vertices.

Otherwise, throw an error if either of them are not.

Checkpoint 3 Passed

**3.**

Let’s verify that we can successfully create an edge between two vertices through the Graph class. Under the Graph class, there are two Vertex instances: atlantaStation and newYorkStation.

Before the trainNetwork is printed, use the trainNetwork’s .addEdge() method to create an edge between the two vertices. We should see Atlanta connect to New York, and New York connect to Atlanta.

Checkpoint 4 Passed

Answer: vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  addEdge(vertex) {

    if (vertex instanceof Vertex) {

      this.edges.push(new Edge(this, vertex));

    } else {

      throw new Error('Edge start and end must both be Vertex');

    }

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor() {

    this.vertices = [];

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  addEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.addEdge(vertexTwo);

      vertexTwo.addEdge(vertexOne);

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph();

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

trainNetwork.addEdge(atlantaStation, newYorkStation);

trainNetwork.print();

module.exports = Graph;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Removing Vertex Connections**

Now that we can connect vertices together, we want to make the Graph more flexible by giving it the ability to remove connections.

We will use the .removeEdge() method to remove any Edge between the given vertex instances.

### Instructions

**1.**

In our Vertex class, create the .removeEdge() method that expects an ending vertex parameter. In order to remove the edge that leads to the given vertex, iterate through its list of edges and filter out the Edge whose end property is strictly equal to the ending vertex.

Checkpoint 2 Passed

**2.**

We’re ready to remove an edge between vertices through our Graph class. In the Graph class, create the .removeEdge() method that removes the edge between two given vertices.

It should expect the vertices as two parameters: vertexOne and vertexTwo. Throw an error if either of them are not Vertex instances. Then use the vertices’ .removeEdge() method to remove the edge between the other vertex. Remember to do this for both vertices.

Checkpoint 3 Passed

**3.**

Let’s verify that we can successfully remove an edge between two vertices through the Graph class. After the edge between Atlanta and New York is added, remove the edges between the two cities. Call the trainNetwork’s .removeEdge() with atlantaStation and newYorkStation.

We should see that the atlantaStation and newYorkStation vertices have no edge connections.

Answer: vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  addEdge(vertex) {

    if (vertex instanceof Vertex) {

      this.edges.push(new Edge(this, vertex));

    } else {

      throw new Error('Edge start and end must both be Vertex');

    }

  }

  removeEdge(vertex) {

    this.edges = this.edges.filter(edge => edge.end !== vertex);

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor() {

    this.vertices = [];

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  addEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.addEdge(vertexTwo);

      vertexTwo.addEdge(vertexOne);

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  removeEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.removeEdge(vertexTwo);

      vertexTwo.removeEdge(vertexOne);

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph();

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

trainNetwork.addEdge(atlantaStation, newYorkStation);

trainNetwork.removeEdge(atlantaStation, newYorkStation);

trainNetwork.print();

module.exports = Graph;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Weighted Graphs**

The current implementation of our Graph class is unweighted, where there is no cost associated with the edge that connects the vertices together. Since we want our Graph to be flexible, we should give the option for weights to be added to the edge when a new edge is created.

### Instructions

**1.**

In the Graph class, add an isWeighted boolean parameter in the constructor for the user to designate that the graph is weighted. It should default to false if no argument is given.

Set the argument to the isWeighted class property.

Checkpoint 2 Passed

**2.**

In the Vertex class, add a second parameter for weight in the .addEdge() method. Pass the argument to the new Edge instance that will be created.

Checkpoint 3 Passed

**3.**

Next, we should feed in the weight argument to the calls to Vertex‘s .addEdge() method from the Graph‘s .addEdge() method if the graph is weighted.

In the Graph class, add a third parameter for weight in the .addEdge() method. Create a variable edgeWeight, and set it to the weight argument if the graph is weighted, otherwise set it to null. Pass edgeWeight to the calls that create edges between the given vertices. Remember to do this for both calls.

Checkpoint 4 Passed

**4.**

Let’s verify that the Graph can add weights to the edges by adding an edge between atlantaStation and newYorkStation. In **Graph.js**, edit trainNetwork to be weighted. Then call the its .addEdge() method and pass in a value of 800 as an argument, to represent the number of miles between the two train stations.

When we call the trainNetwork’s .print() method to print out the resulting graph, we should see that the edge between Atlanta and New York has a value of 800.

Checkpoint

Answer: graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor(isWeighted = false) {

    this.vertices = [];

    this.isWeighted = isWeighted;

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  addEdge(vertexOne, vertexTwo, weight) {

    const edgeWeight = this.isWeighted ? weight : null;

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.addEdge(vertexTwo, edgeWeight);

      vertexTwo.addEdge(vertexOne, edgeWeight);

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  removeEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.removeEdge(vertexTwo);

      vertexTwo.removeEdge(vertexOne);

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph(true);

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

trainNetwork.addEdge(atlantaStation, newYorkStation, 800);

trainNetwork.print();

module.exports = Graph;

vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  addEdge(vertex, weight) {

    if (vertex instanceof Vertex) {

      this.edges.push(new Edge(this, vertex, weight));

    } else {

      throw new Error('Edge start and end must both be Vertex');

    }

  }

  removeEdge(vertex) {

    this.edges = this.edges.filter(edge => edge.end !== vertex);

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Directed Graphs**

So far we have only built out support for undirected graphs. Next, we will focus on expanding our Graph class to be directed, where there does not necessarily have to be edges going in both directions between the vertices, as we have done with undirected graphs.

The main difference between the undirected graph and directed graph is that our undirected graph uses two edges going in opposite directions to indicate that there is a connection between two vertices.

### Instructions

**1.**

When a Graph is first created, we need a way to identify if it will be directed or not. In the constructor, add a parameter. This argument will be a boolean of whether or not the graph is directed. Store the isDirected argument in the Graph’s isDirected property. By default, isDirected should be set to false.

Checkpoint 2 Passed

**2.**

We only want to create one edge that points in one direction between two vertices for directed graphs.

Modify the Graph‘s .addEdge() method to create the edge from vertexTwo to vertexOne only if this isDirected property is false.

Checkpoint 3 Passed

**3.**

Just as we only want to create one edge between vertices in a directed graph, we also want to remove only one edge between vertices.

Modify the Graph‘s .removeEdge() method to remove the edge from vertexTwo to vertexOne if this isDirected property is false.

Checkpoint 4 Passed

**4.**

Finally, let’s modify our train network to only travel in one direction: from New York to Atlanta. Modify the trainNetwork to be unweighted and directed. Pass false for the first argument and true for the second.

We should see only one edge connection going from Atlanta to New York. New York should have no edges going out.

Checkpoint 5 Passed

**5.**

Last, but not least, we should test out our edge removal. Call .removeEdge() on the trainNetwork graph to remove the edge between atlantaStation and newYorkStation.

When we print out the resulting graph, Atlanta and New York should have no connections.

Checkpoint 6 Passed

Answer: graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor(isWeighted = false, isDirected = false) {

    this.vertices = [];

    this.isWeighted = isWeighted;

    this.isDirected = isDirected;

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  addEdge(vertexOne, vertexTwo, weight) {

    const edgeWeight = this.isWeighted ? weight : null;

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.addEdge(vertexTwo, edgeWeight);

      if (!this.isDirected) {

        vertexTwo.addEdge(vertexOne, edgeWeight);

      }

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  removeEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.removeEdge(vertexTwo);

      if (!this.isDirected) {

        vertexTwo.removeEdge(vertexOne);

      }

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  print() {

    const vertexList = this.vertices || [];

    vertexList.forEach(vertex => vertex.print());

  }

}

const trainNetwork = new Graph(false, true);

const atlantaStation = trainNetwork.addVertex('Atlanta');

const newYorkStation = trainNetwork.addVertex('New York');

trainNetwork.addEdge(atlantaStation, newYorkStation);

trainNetwork.removeEdge(atlantaStation, newYorkStation);

trainNetwork.print();

module.exports = Graph;

vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  addEdge(vertex, weight) {

    if (vertex instanceof Vertex) {

      this.edges.push(new Edge(this, vertex, weight));

    } else {

      throw new Error('Edge start and end must both be Vertex');

    }

  }

  removeEdge(vertex) {

    this.edges = this.edges.filter(edge => edge.end !== vertex);

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== undefined ? `${edge.end.data} (${edge.weight})` : edge.end.data) || [];

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

**Graph Review**

Let’s put our Graph class to use and build out our train network. This time we will use it to map out all the train stations and routes that our rail service operates. We will also want to make sure that our routes can track the distance between each station.

For this exercise, we will build out the train network inside **trainNetwork.js**.

### Instructions

**1.**

Start by creating the train network as a weighted and directed Graph instance and assigning it to trainNetwork.

Checkpoint 2 Passed

**2.**

We just got funding to build out 6 train stations. Using the graph’s .addVertex() method, add the following station vertices to our trainNetwork with the names:

* Los Angeles
* San Francisco
* New York
* Atlanta
* Denver
* Calgary

Checkpoint 3 Passed

**3.**

We only want to service the routes our customers will travel the most, so let’s use .addEdge() to add the following route edges to the graph:

* From San Francisco to Los Angeles, which is 400mi
* From Los Angeles to San Francisco, which is 400mi
* From New York to Denver, which is 1800mi
* From Denver to New York, which is 1800mi
* From Calgary to Denver, which is 1000mi
* From Denver to Calgary, which is 1000mi
* From Los Angeles to Atlanta, which is 2100mi
* From Atlanta to Los Angeles, which is 2100mi

Checkpoint 4 Passed

**4.**

Darn! As we were building out our routes, there was a huge snowstorm that hit Calgary and New York. We were able to salvage the route from Denver to New York, but all of the routes to and from Calgary broke down.

Using the graph’s .removeEdge() and .removeVertex() methods, remove the route from New York to Denver, all the routes to and from Calgary, and the Calgary station.

Checkpoint 5 Passed

**5.**

We’re finally all aboard the same page. Print out our final graph and check that we built the following routes:

* San Francisco to and from Los Angeles
* Los Angeles to and from Atlanta
* Denver to New York

This wraps up our graph implementation! There are still some edge (pardon the pun) cases that we have not yet accounted for. If you’re feeling up for it, try to challenge yourself with the following:

* Currently, it is possible to add duplicate edges between two vertices. How will you improve this Graph implementation to avoid adding duplicate edges?
* How would you iterate through a directed graph? What about an undirected graph?
* How would you create a cycle with a directed graph?

Checkpoint 6 Passed

Answer: trainNetwork.js

const Graph = require('./Graph.js');

const trainNetwork = new Graph(true, true);

const laStation = trainNetwork.addVertex('Los Angeles');

const sfStation = trainNetwork.addVertex('San Francisco');

const nyStation = trainNetwork.addVertex('New York');

const atlStation = trainNetwork.addVertex('Atlanta');

const denStation = trainNetwork.addVertex('Denver');

const calStation = trainNetwork.addVertex('Calgary');

trainNetwork.addEdge(sfStation, laStation, 400);

trainNetwork.addEdge(laStation, sfStation, 400);

trainNetwork.addEdge(nyStation, denStation, 1800);

trainNetwork.addEdge(denStation, nyStation, 1800);

trainNetwork.addEdge(calStation, denStation, 1000);

trainNetwork.addEdge(denStation, calStation, 1000);

trainNetwork.addEdge(atlStation, laStation, 2100);

trainNetwork.addEdge(laStation, atlStation, 2100);

trainNetwork.removeEdge(nyStation, denStation);

trainNetwork.removeEdge(calStation, denStation);

trainNetwork.removeEdge(denStation, calStation);

trainNetwork.removeVertex(calStation);

trainNetwork.print();

graph.js

const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');

class Graph {

  constructor(isWeighted = false, isDirected = false) {

    this.vertices = [];

    this.isWeighted = isWeighted;

    this.isDirected = isDirected;

  }

  addVertex(data) {

    const newVertex = new Vertex(data);

    this.vertices.push(newVertex);

    return newVertex;

  }

  removeVertex(vertex) {

    this.vertices = this.vertices.filter(v => v !== vertex);

  }

  addEdge(vertexOne, vertexTwo, weight) {

    const edgeWeight = this.isWeighted ? weight : null;

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.addEdge(vertexTwo, edgeWeight);

      if (!this.isDirected) {

        vertexTwo.addEdge(vertexOne, edgeWeight);

      }

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  removeEdge(vertexOne, vertexTwo) {

    if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {

      vertexOne.removeEdge(vertexTwo);

      if (!this.isDirected) {

        vertexTwo.removeEdge(vertexOne);

      }

    } else {

      throw new Error('Expected Vertex arguments.');

    }

  }

  print() {

    this.vertices.forEach(vertex => vertex.print());

  }

}

module.exports = Graph;

vertex.js

const Edge = require('./Edge.js');

class Vertex {

  constructor(data) {

    this.data = data;

    this.edges = [];

  }

  addEdge(vertex, weight) {

    if (vertex instanceof Vertex) {

      this.edges.push(new Edge(this, vertex, weight));

    } else {

      throw new Error('Edge start and end must both be Vertex');

    }

  }

  removeEdge(vertex) {

    this.edges = this.edges.filter(edge => edge.end !== vertex);

  }

  print() {

    const edgeList = this.edges.map(edge =>

        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data);

    const output = `${this.data} --> ${edgeList.join(', ')}`;

    console.log(output);

  }

}

module.exports = Vertex;

edges.js

class Edge {

  constructor(start, end, weight = null) {

    this.start = start;

    this.end = end;

    this.weight = weight;

  }

}

module.exports = Edge;

# chapter: 40 Introduction: Algorithms

# Recursion: if you called a function or method from within that function or method? That’s called recursion and can actually make your program more efficient. It can be used for problems that can be broken up into multiple steps, often ones that you would otherwise solve iteratively. It will also be used in some of the traversal and sort algorithms

Theory to understand

**Introduction to Recursion**

You’ve heard about a trendy new spot that sells fruit sandwiches. What are fruit sandwiches? You have no idea, but you’re eager to find out!

Sadly, when you arrive at the store, the line is out the door and around the block. Undeterred, you hatch a plan to find out how many people are in line before you.

You tap the person in front of you and ask them how many people are ahead of them. They have no idea, (the line is huge!) so they ask you to wait a moment and tap the person in front of them, repeating this process through the line.

Finally, the second to last person taps the person at the front of the line. Nobody is ahead of them, so they reply “It’s just me so: one person!”. **Then this message is repeated back down the line.**

The next person says “okay, there was one person ahead of me, I’ll add myself… I can tell the person behind me: 2 people in line.”

Each person waiting for a reply:

1. receives the message
2. adds themselves to the count
3. responds to the person tapping them

This chain eventually reaches you with the final number. Your plan was a success!

You executed a recursive strategy. The “function” of asking a person **involved asking a person**. The self-referential logic can seem like it goes on forever, but each question brings you closer to the front of the line where **no more people are asked about the line**.

Your approach had two aspects which are essential to recursive thinking. Break the problem into two possibilities:

1. There’s nobody left in line, don’t ask
2. There’s someone in line, ask them

We repeat Step 2 with a different input which brings us closer to Step 1.

**Recursion Outline**

Recursion is a strategy for solving problems by defining the problem in terms of itself. For example, to **sum the elements of a list** we would take the first element and add it to the **sum of the remaining elements**.

How would we get that sum of remaining elements? Easy! We’d take the first element of the remaining elements and add it to the… Maybe you can understand why recursion can be a tricky subject!

In programming, recursion means a function definition will include an invocation of the function **within its own body.** Here’s a pseudo-code example:

define function, speller  
  if there are no more letters  
    print "all done"  
  print the first letter  
  invoke speller with the given name minus the first letter

If we invoked this function with “Zoe” as the argument, we would see “Z”, “o”, and “e” printed out before “all done”.

We call the function a total of 4 times!

1. function called with “Zoe”
2. function called with “oe”
3. function called with “e”
4. function called with “”

Let’s break the function into three chunks:

   if there are no more letters  
     print "all done"

This section is the *base case*. We are **NOT** invoking the function under this condition. The equivalent base case from the previous exercise was when we had reached the front of the line.

   print the first letter

This section solves **a piece** of the problem. If we want to spell someone’s name, we’ll have to spell **at least** one letter.

   invoke speller with the given name minus the first letter

This section is the *recursive* step, calling the function with arguments which bring us closer to the base case. In this example, we’re reducing the length of the name by a single letter. Eventually, there will be a function call with no letters given as the argument.

For comparison’s sake, here is pseudo-code for an *iterative* approach to the same problem:

define function, speller  
   for each letter in the name argument  
     print the letter  
   print "all done"

**Call Stacks and Execution Frames**

A recursive approach requires the function invoking itself **with different arguments.** How does the computer keep track of the various arguments and different function invocations if it’s the same function definition?

Repeatedly invoking functions may be familiar when it occurs sequentially, but it can be jarring to see this invocation occur **within a function definition**.

Languages make this possible with *call stacks* and *execution contexts*.

Stacks, a data structure, follow a strict protocol for the order data enters and exits the structure: **the last thing to enter is the first thing to leave**.

Your programming language often manages the call stack, which exists outside of any specific function. This call stack tracks the ordering of the different function invocations, so the **last function to enter the call stack is the first function to exit the call stack**.

We can think of execution contexts as the specific values we plug into a function call.

A function which adds two numbers:  
   
Invoking the function with 3 and 4 as arguments...  
execution context:  
X = 3  
Y = 4  
   
Invoking the function with 6 and 2 as arguments...  
execution context:  
X = 6  
Y = 2

Consider a pseudo-code function which sums the integers in an array:

function, sum\_list   
   if list has a single element  
     return that single element  
   otherwise...  
     add first element to value of sum\_list called with every element minus the first

This function will be invoked as many times as there are elements within the list! Let’s step through:

CALL STACK EMPTY  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
Our first function call...  
sum\_list([5, 6, 7])  
   
CALL STACK CONTAINS  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([5, 6, 7])  
with the execution context of a list being [5, 6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
Base case, a list of one element not met.  
We invoke sum\_list with the list of [6, 7]...  
   
CALL STACK CONTAINS  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([6, 7])  
with the execution context of a list being [6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([5, 6, 7])  
with the execution context of a list being [5, 6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
Base case, a list of one element not met.  
We invoke sum\_list with the list of [7]...  
   
CALL STACK CONTAINS  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([7])  
with the execution context of a list being [7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([6, 7])  
with the execution context of a list being [6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([5, 6, 7])  
with the execution context of a list being [5, 6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
We've reached our base case! List is one element.   
We return that one element.  
This return value does two things:  
   
1) "pops" sum\_list([7]) from CALL STACK.  
2) provides a return value for sum\_list([6, 7])  
   
----------------  
CALL STACK CONTAINS  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([6, 7])  
with the execution context of a list being [6, 7]  
RETURN VALUE = 7  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([5, 6, 7])  
with the execution context of a list being [5, 6, 7]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
sum\_list([6, 7]) waits for the return value of sum\_list([7]), which it just received.   
   
sum\_list([6, 7]) has resolved and "popped" from the call stack...  
   
   
----------------  
CALL STACK contains  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
sum\_list([5, 6, 7])  
with the execution context of a list being [5, 6, 7]  
RETURN VALUE = 6 + 7  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
   
sum\_list([5, 6, 7]) waits for the return value of sum\_list([6, 7]), which it just received.   
sum\_list([5, 6, 7]) has resolved and "popped" from the call stack.  
   
   
----------------  
CALL STACK is empty  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
RETURN VALUE = (5 + 6 + 7) = 18

**Base Case and Recursive Step**

Recursion has two fundamental aspects: the base case and the recursive step.

When using iteration, we rely on a counting variable and a boolean condition. For example, when iterating through the values in a list, we would increment the counting variable until it exceeded the length of the dataset.

Recursive functions have a similar concept, which we call the base case. The base case dictates whether the function will recurse, or call itself. Without a base case, it’s the iterative equivalent to writing an infinite loop.

Because we’re using a call stack to track the function calls, your computer will throw an error due to a *stack overflow* if the base case is not sufficient.

The other fundamental aspect of a recursive function is the recursive step. This portion of the function is the step that moves us closer to the base case.

In an iterative function, this is handled by a loop construct that decrements or increments the counting variable which moves the counter closer to a boolean condition, terminating the loop.

In a recursive function, the “counting variable” equivalent is the **argument to the recursive call.** If we’re counting down to 0, for example, our base case would be the function call that receives 0 as an argument. We might design a recursive step that takes the argument passed in, decrements it by one, and calls the function again with the decremented argument. In this way, we would be moving towards 0 as our base case.

Analyzing the Big O runtime of a recursive function is very similar to analyzing an iterative function. Substitute iterations of a loop with recursive calls.

For example, if we loop through once for each element printing the value, we have a O(N) or linear runtime. Similarly, if we have a single recursive call for each element in the original function call, we have a O(N) or linear runtime.

Exercise:

**Introduction**

Recursion is a powerful tool for solving problems that require the execution of a similar action multiple times until a certain condition is met. For many problems, a recursive solution will result in fewer lines of code and will be easier to comprehend than a solution that uses a for or while loop.

You may find that recursion is a difficult concept to wrap your head around at first. That’s fine! This lesson is meant as an introduction. As you see more examples, you will start to feel comfortable with the concept.

In this lesson, you will learn about recursion while implementing a function that returns the factorial of a number. Factorial is the product of an integer and all positive numbers less than it.

Let’s consider 4 factorial:

4! = 4 \times 3 \times 2 \times 1 = 244!=4×3×2×1=24

Four factorial is equal to the product of 4 x 3 x 2 x 1, which is 24. The exclamation mark denotes that the number 4 is being factorialized.

1! and 0! are both valid base cases of factorial. The factorial product of both numbers is 1.

Before we dive into recursion, you will consider how factorial is implemented with an iterative approach.

### Instructions

**1.**

**index.js** includes a function called iterativeFactorial(). The function accepts an integer as an argument, and returns the factorial of it.

Take a look at how we implemented the function. Run the code when you’re ready to move to the next checkpoint.

Checkpoint 2 Passed

**2.**

Set a constant named fourFactorial equal to the value returned from iterativeFactorial(), with 4 as the argument.

Checkpoint 3 Passed

**3.**

Log the value saved to fourFactorial to the console.

Checkpoint 4 Passed

Answer: index.js

const iterativeFactorial = (n) => {

  let result = 1;

  while(n > 0) {

    result \*= n;

    n -= 1;

  }

  return result;

}

// Set fourFactorial

const fourFactorial = iterativeFactorial(3);

console.log(fourFactorial);

module.exports = {

  iterativeFactorial

};

**Recursion**

So, what is recursion?

Recursion is a computational approach where a function calls itself from within its body. Programmers use recursion when they need to perform the similar action multiple times in a row until it reaches a predefined stopping point, also known as a base case.

Let’s think about this in the context of our factorial example. Below is the beginnings of a recursive implementation of factorial. This code is all in **index.js**, to the right.

const recursiveFactorial = (n) => {  
  if (condition){  
    console.log(`Execution context: ${n}`);  
   
    recursiveFactorial(n - 1);  
  }  
}

Within the recursiveFactorial() function, we want to check whether a condition is met. If it is, then we print the value of n and return a call to recursiveFactorial(n - 1).

Can you think of a condition that will result in the following response when we call recursiveFactorial(4)?

Execution context: 4  
Execution context: 3  
Execution context: 2  
Execution context: 1  
Execution context: 0

The correct answer is n > 0. At this point, we have the beginnings of a recursive function, but we’re still not returning anything.

### Instructions

**1.**

Change the condition in the if statement to something that will prevent recursiveFactorial() from calling itself if n is less than 1.

Checkpoint 2 Passed

Answer: index.js

const recursiveFactorial = (n) => {

  if (n > 0) {

    console.log(`Execution context: ${n}`);

    recursiveFactorial(n - 1);

  }

}

const recursiveSolution = recursiveFactorial(4);

console.log(recursiveSolution);

module.exports = {

  recursiveFactorial

};

**Recursive Case**

In the last exercise, you created a condition (n > 0 or n >= 1). This condition is important, because it defines whether or not recursiveFactorial() calls itself. We call this if block the recursive case.

In recursion, the recursive case is the condition under which a function calls itself. We call this the recursive case because, as mentioned last exercise, recursion is defined as a process when a function calls itself.

At the end of last exercise, your output should have looked like:

Execution context: 4  
Execution context: 3  
Execution context: 2  
Execution context: 1  
undefined

At this point, there are a couple of shortcomings in the implementation that are worth mentioning:

* Calculating the product of the numbers – while we do access all of the numbers that need to be multiplied, we do not calculate their product.
* recursiveSolution is set to undefined – the value set to recursiveSolution (see **index.js** to the right) is undefined, because we never returned anything from recursiveFactorial().

### Instructions

**1.**

In your function, return the product of n and your call to recursiveFactorial().

After you run your code, you should see that the value saved to recursiveSolution has changed. Is it what you expect?

Checkpoint 2 Passed

Answer: index.js

const recursiveFactorial = (n) => {

  if (n > 0){

    console.log(`Execution context: ${n}`);

    return recursiveFactorial(n - 1) \* n;

  }

}

const recursiveSolution = recursiveFactorial(4);

console.log(recursiveSolution);

module.exports = {

  recursiveFactorial

};

**Base Case**

The solution to the last exercise resulted in the following output:

Execution context: 4  
Execution context: 3  
Execution context: 2  
Execution context: 1  
NaN

Notice, the value saved to recursiveSolution changed from undefined to NaN (not a number).

Why is *recursiveSolution* not a number? The short answer: we didn’t define a base case. To understand the need for a base case, it’s worth discussing the call stack that JavaScript creates when you call recursiveFactorial().

If you were to call:

recursiveSolution = recursiveFactorial(3)

JavaScript would create a call stack with the following events:

1. recursiveFactorial(3) = 3 \* recursiveFactorial(2)
2. recursiveFactorial(2) = 2 \* recursiveFactorial(1)
3. recursiveFactorial(1) = 1 \* recursiveFactorial(0)

The return value associated with each function call depends on the value returned by the n - 1 context. Because the current implementation does not return a number for recursiveFactorial(0), the result of (3) is NaN. This leads to an NaN solution for each of the contexts above it.

We need a base case to address the NaN returned from the n === 0 context. The factorial function should return a number when n === 0.

### Instructions

**1.**

We set recursiveSolution equal to the value returned from recursiveFactorial() with 0 as the argument.

Run the code. You should see undefined in the terminal.

Checkpoint 2 Passed

**2.**

Inside recursiveFactorial(), add an if statement that returns 1 when n is equal to 0.

Checkpoint 3 Passed

**3.**

Set recursiveSolution equal to the value returned from recursiveFactorial() with 5 as the argument.

Checkpoint 4 Passed

Answer: index.js

const recursiveFactorial = (n) => {

  // Add a condition below

  if (n === 0) {

    return 1;

  }

  if (n > 0){

    console.log(`Execution context: ${n}`);

    return recursiveFactorial(n - 1) \* n;

  }

}

const recursiveSolution = recursiveFactorial(5);

console.log(recursiveSolution);

module.exports = {

  recursiveFactorial

};

**Review**

Throughout this lesson, you learned about recursion as you coded a factorial function. While every recursive problem is a little different, the following features will always be required:

* Recursive case – the conditions under which the function will perform an action and call itself.
* Base case – the conditions under which the function returns a value without making any additional calls to itself.

In this example, we started by considering the recursive case. With some problems it may be easier to start with the base case. Regardless, when you are dealing with a recursive problem, start by considering each of these cases.

### Instructions

In **index.js**, we included both the iterative and recursive solutions to factorial. Both approaches work equally well for this problem.

As you learn more about recursion, you may find the syntax to be more readable and easier to understand when addressing certain problems. Consider it another tool in your toolbox as you approach increasingly challenging programming problems.

const recursiveFactorial = (n) => {

  if (n === 0) {

    return 1;

  } else if (n > 0) {

    return n \* recursiveFactorial(n - 1);

  }

}

const iterativeFactorial = (n) => {

  result = 1;

  while(n > 0) {

    result \*= n;

    n -= 1;

  }

  return result;

}

module.exports = {

  recursiveFactorial,

  iterativeFactorial

};

**RECURSIVE VS. ITERATIVE TRAVERSAL**

**Introduction**

In this lesson, you will learn how to implement a recursive solution to a linked list search. The method accepts a value as input and recursively checks each node in the linked list, until the node of interest is found. If it is found, the method should return the node. Otherwise, it should return null.

Before you begin, let’s take a look at how we can search for an element in a linked list using an iterative approach. The code below is taken from the LinkedList() class in **LinkedList.js**.

findNodeIteratively(data) {  
  let currentNode = this.head;  
  while (currentNode !== null) {  
    if (currentNode.data === data) {  
      return currentNode;  
    }  
    currentNode = currentNode.next;  
  }  
  return null;  
}

The method starts at the head of the linked list and checks if the input data is equal to the data parameter at the head. The method continues to iterate through the linked list until the node is found or the end of the list is reached.

### Instructions

**1.**

In **index.js**, use .findNodeIteratively() to find the Node in myList with data equal to 'Node 3'.

Save the returned value to foundNode, then log it to the console.

Checkpoint 2 Passed

Answer: index.js

const Node = require('./Node');

const LinkedList = require('./LinkedList');

const myList = new LinkedList();

myList.addToHead('Node 1');

myList.addToHead('Node 2');

myList.addToHead('Node 3');

myList.addToHead('Node 4');

// Add code below

const foundNode = myList.findNodeIteratively('Node 3');

console.log(foundNode);

node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

Linkedlist.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

  }

}

module.exports = LinkedList;

**Base Case**

Before we consider the base and recursive cases, let’s think about the two parameters required to traverse a linked list recursively:

* data – the first parameter. This is the value of the Node that is being searched for in the linked list.
* currentNode – the second parameter. This is the current node in the linked list. During each recursive call, the function will pass the next node as this argument.

class LinkedList {  
   
  findNodeRecursively(data, currentNode = this.head) {  
   // Some code    
  }  
}

Notice, we added this.head as the default argument for currentNode. This is useful because, if you call findNodeRecursively() with only a data argument, the method will traverse the entire linked list beginning from its head.

Now let’s consider the base case for our linked list. We should return a value under the following two cases:

* If the method finds a node with the matching value, it should return the node.
* If the method reaches the end of the list, it should return null.

### Instructions

**1.**

Return null when .findNodeRecursively() reaches the end of the linked list.

Checkpoint 2 Passed

**2.**

In .findNodeRecursively(), add an else if statement that returns the currentNode if the node’s data attribute is equal to the input data argument.

Checkpoint 3 Passed

**3.**

Add the following code to **index.js**:

const myNodeRecursive = myList.findNodeRecursively('Node 4');  
console.log(myNodeRecursive);

This code should find the Node with a data argument equal to 'Node 4' and log it to the console.

Checkpoint 4 Passed

**4.**

Change the call to .findNodeRecursively() so it searches for 'Node 3'. Run the code.

This will print undefined, because we have not set our recursive case.

Checkpoint 5 Passed

Answer: linedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    // Add base cases below

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    }

  }

}

module.exports = LinkedList;

node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

index.js

const Node = require('./Node');

const LinkedList = require('./LinkedList');

const myList = new LinkedList();

myList.addToHead('Node 1');

myList.addToHead('Node 2');

myList.addToHead('Node 3');

myList.addToHead('Node 4');

// Add checkpoint 3 code below:

const myNodeRecursive = myList.findNodeRecursively('Node 4');

console.log(myNodeRecursive);

**Recursive Case**

Now it’s time to add a recursive case. The recursive case should execute when the node has not been found and the end of the list has not been reached.

Because you’ve covered both of the base cases, you can use an else statement to call your recursive case.

### Instructions

**1.**

Add an else block to .findNodeRecursively() that returns a call to .findNodeRecursively().

Pass data and the next node as arguments.

Checkpoint 2 Passed

**2.**

In **index.js**, change the argument passed to .findNodeIteratively() to 'Node 2'.

Checkpoint 3 Passed

Answer: linkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

index.jsconst Node = require('./Node');

const LinkedList = require('./LinkedList');

const myList = new LinkedList();

myList.addToHead('Node 1');

myList.addToHead('Node 2');

myList.addToHead('Node 3');

myList.addToHead('Node 4');

// Add checkpoint 2 code below:

const myNodeRecursive = myList.findNodeIteratively('Node 2');

console.log(myNodeRecursive);

**Review**

In this lesson, you learned how to implement a recursive solution to a linked list search. The solution includes the following cases:

* *Base case 1* – return the current node if it matches the data argument.
* *Base case 2* – return null if the end of the linked list is reached.
* *Recursive Case* – return a call to .findNodeRecursively() with the next node as an argument.

The recursive approach laid out in this lesson is similar to implementations for traversing other data structures, like trees and directories. This is an important insight to keep in mind as you encounter more recursive implementations.

Answer : linkedlist.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

  findNodeIteratively(data) {

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        return currentNode;

      }

      currentNode = currentNode.next;

    }

    return null;

  }

  findNodeRecursively(data, currentNode = this.head) {

    if (currentNode === null) {

      return null;

    } else if (currentNode.data === data) {

      return currentNode;

    } else {

      return this.findNodeRecursively(data, currentNode.next);

    }

  }

}

module.exports = LinkedList;

node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

index.js

const Node = require('./Node');

const LinkedList = require('./LinkedList');

const myList = new LinkedList();

myList.addToHead('Node 1');

myList.addToHead('Node 2');

myList.addToHead('Node 3');

myList.addToHead('Node 4');

// Add checkpoint 2 code below:

const myNodeRecursive = myList.findNodeIteratively('Node 2');

console.log(myNodeRecursive);

# Why Asymptotic Notation?

Programming is also done in many different languages, how do we account for that in the runtime? We need a general way to define a program’s runtime across these variable factors. We do this with **Asymptotic Notation**.

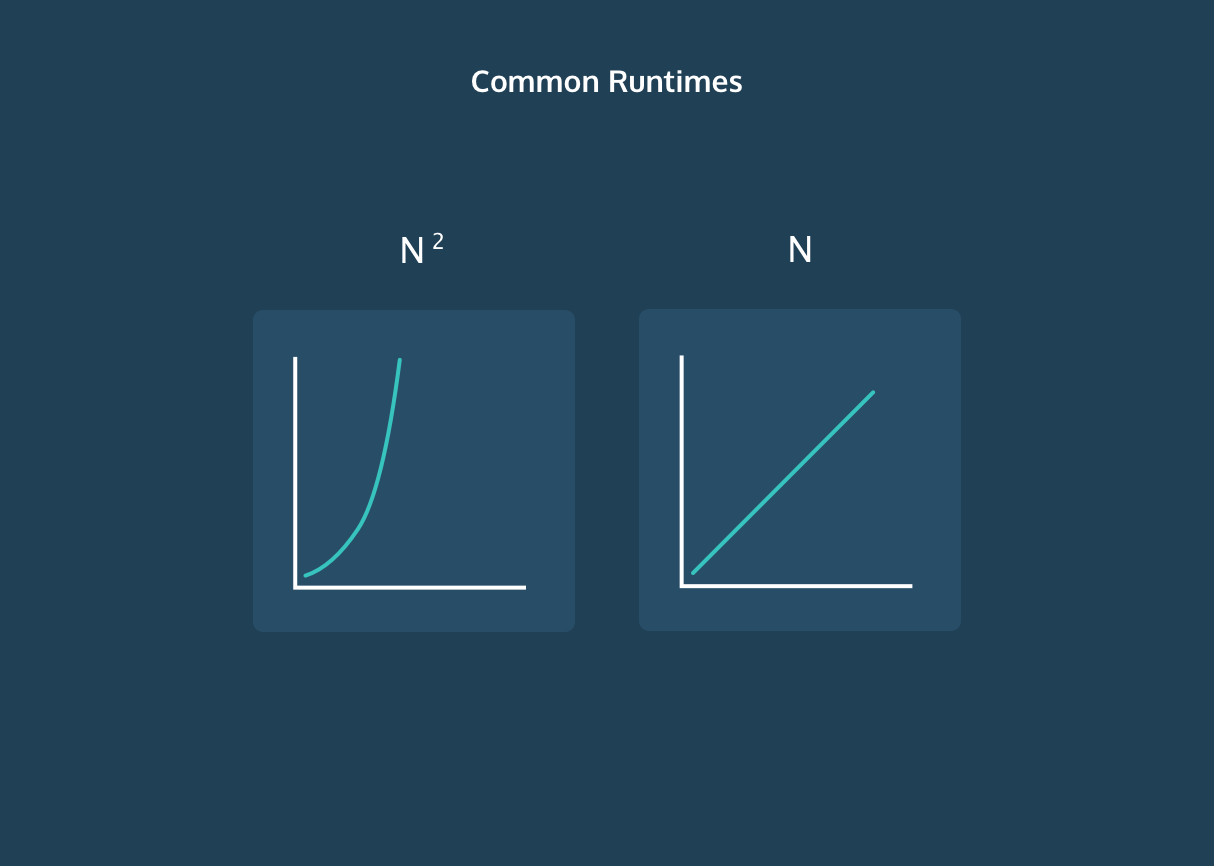
With asymptotic notation, we calculate a program’s runtime by looking at how many instructions the computer has to perform based on the size of the program’s input. For example, if I were calculating the maximum element in a collection, I would need to examine each element in the collection. That examining step is the same regardless of the language used, or the CPU that’s performing the calculation. In asymptotic notation, we define the size of the input as N. I may be looking through a collection of 10 elements, or 100 elements, but we only need to know how many steps are performed *relative to the input* so N is used in place of a specific number. If there is a second input, we may define the size of that input as M.

There are varieties of asymptotic notation that focus on different concerns. Some will communicate the best case scenario for a program. For example, if we were searching for a value within a collection, the best case would be if we found that element in the first place we looked. Another type will focus on the worst case scenario, such as if we searched for a value, looked in the entire dataset and did not find it. Typically programmers will focus on the worst case scenario so there is an upper bound of runtime to communicate. It’s a way of saying “things may get this bad, or slow, but they won’t get worse!”

For example, a program that takes 12 nanoseconds on one computer could take 45 milliseconds on another. Therefore, we need a more general way to gauge a program’s runtime. We do this with Asymptotic Notation.

Instead of timing a program, through asymptotic notation, we can calculate a program’s runtime by looking at how many instructions the computer has to perform based on the size of the program’s input: N.

For instance, a program that has input of size N may tell the computer to run 5N2+3N+2 instructions. (We will get into how we get this kind of expression in future exercises.) Nevertheless, this is still a fairly messy and large expression. For asymptotic notation, we drop all of our constants (the numbers) because as N becomes extremely large, the constants will make minute differences. After changing our constants, we have N2+N. If we take each of these terms in the expression and graph them, we see that the N2 term grows faster than the N term.



For example, when N is 1000:

* the N2 term is 1,000,000
* the N term is 1,000

As you can see, the N2 term is much more significant than the N term. When N is larger than 1000, the difference becomes even more significant. Because the difference is so enormous, we don’t even need to consider the N term when calculating the runtime. Thus, for this program, we would describe the runtime in terms of N2. There are three different ways we could describe the runtime of this program: big Theta or Θ(N2), big O or O(N2), big Omega or Ω(N2). The difference between the three and when to use which one will be detailed in the next exercises.

You may see the term **execution count** used in evaluating algorithms. Execution count is more precise than Big O notation. The following method, addUpTo(), depending on how we count the number of operations, can be as low as 2N or as high as 5N + 2

public class Main() {   
  void int addUpTo(int n) {  
    int total = 0;  
    for (int i = 1; i <= n; i++) {  
      total += i;  
    }  
  return total;  
  }   
}

Determining execution count can increase in difficulty as our algorithms become even more sophisticated!

But regardless of the execution count, the number of operations grows roughly proportionally with n. If n doubles, the number of operations will also roughly double.

Big O Notation is a way to formalize fuzzy counting. It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow. As we will see, Big O doesn’t focus on the details, only the trends

### IN SHORTS

### Asymptotic Notation

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω). big-Θ is used when the running time is the same for all cases, big-O for the worst case running time, and big-Ω for the best case running time.

**Big Theta (Θ)**

The first subtype of asymptotic notation we will explore is big Theta (denoted by Θ). We use big Theta when a program has only one case in term of runtime. But what exactly does that mean? Take a look at the pseudocode for a function that prints the values in a list below:

Function with input that is a list of size N:  
   For each value in list:  
    Print the value

The number of instructions the computer has to perform is based on how many iterations the loop will do because if the loop does more iterations, then the computer will perform instructions. Now, let’s see how many iterations the loop will do dependent on the value of N.

| **Size of List** | **–vs.–** | **Number of Iterations** |
| --- | --- | --- |
| 1 |  | 1 |
| 2 |  | 2 |
| 3 |  | 3 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| N |  | N |

As we can see in every case, with a list of size N, the program has a runtime of N because the program has to print a value N times. Thus, we would say the runtime is Θ(N).

Let’s look at a more complicated example. In the following pseudocode program, the function takes in an integer, N, and counts the number of times it takes for N to be divided by 2 until N reaches 1.

Function that has integer input N:  
    Set a count variable to 0  
    Loop while N is not equal to 1:  
        Increment count  
        N = N/2  
    Return count

Now, let’s see how many iterations the loop will perform based on N.

| **N** | **–vs.–** | **Number of Iterations** |
| --- | --- | --- |
| 1 |  | 0 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| 2 |  | 1 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| 4 |  | 2 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| 8 |  | 3 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| 16 |  | 4 |
| **.** |  | **.** |
| **.** |  | **.** |
| **.** |  | **.** |
| N |  | log2N |

As we can see, in every case, with an integer N, the loop will iterate log2(N) times. However, because we drop constants in asymptotic notation, we would say that the runtime of this program is Θ(log N).

But what happens when there are multiple runtime cases for a single program? We will learn about that in a future exercise.

**Common Runtimes**

* **Θ(1)**. This is constant runtime. This is the runtime when a program will always do the same thing regardless of the input. For instance, a program that only prints “hello, world” runs in Θ(1) because the program will always just print “hello, world”.
* **Θ(log N)**. This is logarithmic runtime. You will see this runtime in search algorithms.
* **Θ(N)**. This is linear runtime. You will often see this when you have to iterate through an entire dataset.
* **Θ(N\*logN)**. You will see this runtime in sorting algorithms.
* **Θ(N2)**. This is an example of a polynomial runtime. You will see this runtime when you have to search through a two-dimensional dataset (like a matrix) or nested loops.
* **Θ(2N)**. This is exponential runtime. You will often see this runtime in recursive algorithms (Don’t worry if you don’t know what that is yet!).
* **Θ(N!)**. This is factorial runtime. You will often see this runtime when you have to generate all of the different permutations of something. For instance, a program that generates all the different ways to order the letters “abcd” would run in this runtime.

## Image

**Big Omega (Ω) and Big O (O)**

Sometimes, a program may have a different runtime for the best case and worst case. For instance, a program could have a best case runtime of Θ(1) and a worst case of Θ(N). We use a different notation when this is the case. We use big Omega or Ω to describe the best case and big O or O to describe the worst case. Take a look at the following pseudocode that returns True if 12 is in the list and False otherwise:

Function with input that is a list of size N:  
    For each value in the list:  
        If value is equal to 12:  
            Return True  
    Return False

How many times will the loop iterate? Let’s take a list of size 1000. If the first value in the list was 12, then the loop would only iterate once. However, if 12 wasn’t in the list at all, the loop would iterate 1000 times. If the input was a list of size N, the loop could iterate anywhere from 1 to N times depending on where 12 is in the list (or if it’s in the list at all). Thus, in the best case, it has a constant runtime and in the worst case it has a linear runtime.

There are many ways we could describe the runtime of this program:

* This program has a best case runtime of Θ(1).
* This program has a worst case runtime of Θ(N).
* This program has a runtime of Ω(1).
* This program has a runtime O(N).

You may be tempted to say the following:

* This program has a runtime of Θ(N).

However, this is not true because the program does not have a linear runtime in every case, only the worst case.

In fact, when describing runtime, people typically discuss the worst case because you should always prepare for the worst case scenario! **Often times, in technical interviews, they will only ask you for the big O of a program.**

Great! Now you know the different types of asymptotic notation and when to use which one! Now, let’s delve into more complex program runtimes!

**Adding Runtimes**

Sometimes, a program has so much going on that it’s hard to find the runtime of it. Take a look at the pseudocode program that first prints all the positive values up to N and then returns the number of times it takes to divide N by 2 until N is 1.

Function that takes a positive integer N:  
    Set a variable i equal to 1  
    Loop until i is equal to N:  
        Print i  
        Increment i  
   
    Set a count variable to 0  
    Loop while N is not equal to 1:  
        Increment count  
        N = N/2  
    Return count

Rather than look at this program all at once, let’s divide into two chunks: the first loop and the second loop.

* In the first loop, we iterate until we reach N. Thus the runtime of the first loop is Θ(N).
* However, the second loop, as demonstrated in a previous exercise, runs in Θ(log N).

Now, we can add the runtimes together, so the runtime is Θ(N) + Θ(log N).

However, when analyzing the runtime of a program, we only care about the slowest part of the program, and because Θ(N) is slower than Θ(log N), we would actually just say the runtime of this program is Θ(N). **It is also appropriate to say the runtime is O(N) because if it runs in Θ(N) for every case, then it also runs in Θ(N) for the worst case. Most of the time people will just use big O notation.**

* We use asymptotic notation to describe the runtime of a program. The three types of asymptotic notation are big Theta, big Omega, and big O.
* We use big Theta (Θ) to describe the runtime if the runtime of the program is the same in every case.
* The different common runtimes from fastest to slowest are: Θ(1), Θ(log N), Θ(N), Θ(N log N), Θ(N2), Θ(2N), Θ(N!).
* We use big Omega (Ω) to describe the best-case running time of a program.
* We use big O (O) to describe the worst-case running time of a program.
* **We typically describe a program’s running time in terms of big O.**
* When finding the runtime of a program with multiple steps, you can divide the program into different sections and add the runtimes of the various sections. You can then take the slowest runtime and use that runtime to describe the entire program.
* When analyzing the runtime of a program, we care about which part of the program is the slowest.

# Asymptotic Notation: JavaScript

## Analyzing Runtimes

Now that you’ve started learning how to use asymptotic notation to measure the runtime of a function, let’s practice with JavaScript!

When analyzing the runtime of a function, it’s necessary to check the number of iterations the loop will perform based on the size of the input.

The **divideByTwo()** function below takes in a positive integer of size **n**, and returns the number of times **n** must be divided by **2** until it reaches **1**.

We can analyze the runtime of this function by counting the number of iterations the **while** loop will perform based on the size of the input (**n**).

Code Challenge

Run the following code on multiple inputs to see how the number of iterations the **while** loop will perform changes.

function divideByTwo(n) {

  let countIterations = 0;

  while (n > 1) {

    n = n / 2;

    countIterations++;

  }

  return countIterations;

}

console.log(divideByTwo(1));

console.log(divideByTwo(2));

console.log(divideByTwo(4));

console.log(divideByTwo(8));

console.log(divideByTwo(16));

console.log(divideByTwo(32));

console.log(divideByTwo(64));

ANSWER:

0

1

2

3

4

5

6

You got it!

Do you notice a pattern forming? With **n** being divided by 2 each iteration, we can use that to establish a big O runtime.

Which of the following is the big O runtime of this algorithm? AND THE ANSWER: log n

**divideByTwo()** has a big O runtime of **log n** because the function divides **n** by two every iteration, and terminates when **n** is 1. **countIterations** counts how many times the **while** loop runs, and you can see in the output that it is log2(**n**). Since we drop constants for asymptotic notation, the big O runtime is just **log n**.

## Finding the Maximum Value in a Linked List

Now that we can analyze the runtime of a function, let’s see take a look at the runtime of data structures.

We often search through data structures to find a specific value. In this exercise, you will complete a function that finds the maximum value of a linked list, and you will also analyze the runtime of your function.

The function, **findMax()**, takes in **list** as an input. The function should return the maximum value in the linked list.

function findMax(list) {  
  let current = list.head;  
  let max = current.data;  
  while (current.getNextNode() !== null) {  
    current = current.getNextNode();  
    let val = current.data;  
    if (val > max) {  
      max = val;  
    }  
  }  
  return max;  
}

Since you only traversed the list once to find the maximum value, what is the big O runtime of the **findMax()** function? ANSWER: n (Yes! The big O runtime is **n** since you iterate through the list one time.

)

## Sorting a Linked List

We also often sort data structures in order to organize the values stored in them. In this exercise, you will sort a linked list from the smallest value to the largest value.

There are many ways to sort a linked list, but one way is as follows:

1. Instantiate a new linked list
2. Find the maximum value of the linked list input
3. Insert the maximum to the beginning of the new linked list
4. Remove the maximum value from the linked list input
5. Repeat steps 2-4 until the linked list input is empty
6. Return the new linked list

Fill in the Code

Fill in the **sortLinkedList()** function such that you return a new linked list that is sorted from smallest to largest. The function uses **findMax()** from above to return the largest element in the list.

function sortLinkedList(list) {  
  let newList = new LinkedList();  
  while (list.head !== null) {  
    let currentMax = findMax(list);  
    list.remove(currentMax);  
    newList.addToHead(currentMax);  
  }  
  return newList;  
}

What is the big O runtime? Remember that **findMax()** is called within the **sortLinkedList()** function. (For the sake of this function, assume that **.remove()** has a big O runtime of **1**.) ANSWER: n^2 (Yes! Since there are nested **while** loops (one in **findMax()** and one in **sortLinkedList()**), the big O runtime is **n^2**.

)

## Stack Runtimes vs Queue Runtimes

In addition to analyzing the runtimes of various data structures, it is also important to compare the runtimes of different data structures.

We will compare the runtimes of retrieving the first value added to a queue to the runtime of retrieving the first value added to a stack.

### Removing the First Value Added to a Queue

A queue is a FIFO (first in, first out) data structure, which means that the first element added to it, will always be the first element removed from it. Removing this element does not require you to iterate through the queue.

### Removing the First Value Added to a Stack

On the other hand, a stack is a FILO (first in, last out) data structure. This means that the first element added will be the last element removed. Removing this element will require you to iterate through the stack, all the way to the bottom.

The big O runtime of removing the first element added to a Queue is **\_\_\_\_\_**, and the big O runtime of removing the first element added to a Stack is **\_\_\_\_\_**. ANSWER: o(1), o(n)

Yes! Since the element will be at the head of the queue, removing it is one step, so it is **O(1)**. On the other hand, since the element will be at the bottom of the stack, removing it requires iterating through the whole stack, so it is **O(n)**.Show less

While finding the first value added to a queue has a better big O runtime than doing so in a stack, consider finding the last value added. In a queue, we will have to iterate through the entire queue to retrieve the element at the end. This will be a big O runtime of **O(n)**. On the other hand, the last value added to a stack is the value at the top of the stack, so removing it will just be a big O runtime of **O(1)**.

## Hash Map Runtimes vs Linked List Runtimes

Similarly, let’s compare the runtimes of searching for a particular element in a linked list and in a hash map.

### Retrieving an Element from a Linked List

To find an element in a linked list, we will have to search through the entire list to see if the element is there. Refer to the **findMax()** function we looked at above for an example. Iterating through the list means that this process has a big O runtime of **O(n)**.

### Retrieving an Element from a Hash Map

Retrieving an element from a hash map is more efficient, due to its structure. Hash maps store information using key-value pairs, which means that every value is linked to a unique key. In order to find the value from the key, it uses the hash function, which has a big O runtime of **O(1)**. If you don’t have to search through the entire data structure, retrieving an element from a hash map is faster than retrieving an element from a linked list.

However, there is the possibility that the element you are looking for is not at the spot that you expect it to be. This happens when two keys have the same hash. There are a few ways hash maps resolve this issue, including separate chaining and open addressing.

#### Separate Chaining

One way to solve hash map collisions is to create a linked list at the array index where the collision occurred. All elements that hash to the same index will be in that list. This means that to find an element in a hash map that uses separate chaining, you must first find the correct index, and then search through the list at that index (if there is more than one element).

Multiple Choice Question

Given the multiple steps required to retrieve an element from a hash map that uses separate chaining, what is the big O runtime of that retrieval? AND THE ANSWER IS n (Yes! The worst case would be that all elements in the hash map hashed to the same index and are in one linked list with the element you’re looking for at the end of the list. To find it, you would have to iterate through the list, which means the big O runtime is **O(n)**.Show less

)

#### Open Addressing

Another way to solve hash map collisions is to simply move down the array until you find an open index, and place the element there. This is a type of open addressing that is called linear probing. When retrieving an element from a hash map that uses linear probing, the worst case would be if the element hashes to the first index, but is actually at the last index. Since you would have to search through the entire array, the big O runtime for retrieving an element from this kind of hash map is **O(n)**.

# Space Complexity

Asymptotic notation is often used to describe the runtime of a program or algorithm, but it can also be used to describe the space, or memory, that a program or algorithm will need. Think about a simple function that takes in two numbers and returns their sum:

function addNumbers(a, b) {  
  return a + b;  
}

This function has a space complexity of **O(1)**, because the amount of space it needs will not change based on the input. While this function also has a constant runtime of **O(1)**, most functions do not have matching space and time complexities:

function simpleLoop(inputArray) {  
  for (let i = 0; i < inputArray.length; i++) {   
    console.log(i);  
  }  
}

As we know, a simple **for** loop that goes through every element in an array of size n has a linear runtime of **O(n)**. However, this function takes **O(1)** space since no new variables are being created and therefore no more space must be allocated.

A recursive function that is passed the same array or object in each call doesn’t add to the space complexity if the array or object is passed by reference (which it is in JavaScript).

Like with time complexity, space complexity denotes space growth in relation to the input size. It’s also important to note that space complexity usually refers to any additional space that will be needed, and doesn’t count the space of the input. So a function could have 10 arrays passed into it, but if all it does inside is print **'Hello World!'**, then it still takes **O(1)** space.

Multiple Choice Question

Consider the **doubleArray()** and **findMin()** functions. Both functions have big O runtimes of **O(n)**, but what are their space complexities?

function doubleArray(inputArray) { // Returns an array that is the double of the input array  
  const doubledArray = [];  
  for (let i = 0; i < inputArray.length; i++) {  
    doubledArray[i] = 2 \* inputArray[i];  
  }  
  return doubledArray;  
}  
  
function findMin(inputArray) { // Returns the smallest element in the array  
  let min = inputArray[0];  
  for (let i = 0; i < inputArray.length; i++) {  
    if (inputArray[i] < min) {  
      min = inputArray[i];  
    }  
  }  
  return min;  
}

Yes! **doubleArray()** creates a new array that matches the size of the input array, so the space needed for this function will change as the size of the input array changes. **findMin()** only creates one new variable regardless of the input, so its size is constant.Show less …doubleArray() has a space complexity of o(n) and findMin has a space complexity of o(1)…

Space complexity is important to consider alongside time complexity when comparing data structures and algorithms. While two functions may have very similar runtimes, one could use less space. Consider the **doubleArray()** function from above. It has a runtime of **O(n)**, and takes **O(n)** space. Could we optimize it to have a better space complexity?

function doubleInPlace(inputArray) {   
  for (let i = 0; i < inputArray.length; i++) {  
    inputArray[i] \*= 2;  
  }  
  return inputArray;  
}

**doubleInPlace()** does the same thing as **doubleArray()** and in the same amount of time, but only takes **O(1)** space, simply because it doesn’t create a new array. As you move forward, remember that just because a program has the best runtime possible, doesn’t mean it can’t still be optimized.

# Sorting Algorithms

Binary search is efficient when given a pre-sorted data set, but what are the most efficient ways to actually sort data sets? There are many different sorting algorithms, but you are going to focus on three of the most common: bubble sort, merge sort, and quicksort. Each has pros and cons when it comes to their efficiency, as you’ll see when you build them. These are not all the possible sorts, but they are a good sampling of the most common sorting algorithms.

**Bubble Sort Introduction**

Bubble sort is an introductory sorting algorithm that iterates through a list and compares pairings of adjacent elements.

According to the sorting criteria, the algorithm swaps elements to shift elements towards the beginning or end of the list.

By default, a list is sorted if for any element e and position 1 through N:

e1 <= e2 <= e3 … eN, where N is the number of elements in the list.

For example, bubble sort transforms a list:

[5, 2, 9, 1, 5]

to an ascending order, from lowest to highest:

[1, 2, 5, 5, 9]

We implement the algorithm with two loops.

The first loop iterates as **long as the list is unsorted** and we assume it’s unsorted to start.

Within this loop, another iteration moves through the list. For each pairing, the algorithm asks:

In comparison, is the first element larger than the second element?

If it is, we swap the position of the elements. The larger element is now at a greater index than the smaller element.

When a swap is made, we know the list is still unsorted. The outer loop will run again when the inner loop concludes.

The process repeats until the largest element makes its way to the last index of the list. The outer loop runs until no swaps are made within the inner loop.

**Bubble Sort**

As mentioned, the bubble sort algorithm swaps elements if the element on the left is larger than the one on the right.

How does this algorithm swap these elements in practice?

Let’s say we have the two values stored at the following indices index\_1 and index\_2. How would we swap these two elements within the list?

It is tempting to write code like:

list[index\_1] = list[index\_2]  
list[index\_2] = list[index\_1]

However, if we do this, we lose the original value at index\_1. The element gets replaced by the value at index\_2. Both indices end up with the value at index\_2.

Programming languages have different ways of avoiding this issue. In some languages, we create a temporary variable which holds one element during the swap:

temp = list[index\_1]  
list[index\_1] = list[index\_2]  
list[index\_2] = temp

The GIF illustrates this code snippet.

Other languages provide multiple assignment which removes the need for a temporary variable.

list[index\_1], list[index\_2] = list[index\_2], list[index\_1]

**Algorithm Analysis**

Given a moderately unsorted data-set, bubble sort requires multiple passes through the input before producing a sorted list. Each pass through the list will place the next largest value in its proper place.

We are performing n-1 comparisons for our inner loop. Then, we must go through the list n times in order to ensure that each item in our list has been placed in its proper order.

The n signifies the number of elements in the list. In a worst case scenario, the inner loop does n-1 comparisons for each n element in the list.

Therefore we calculate the algorithm’s efficiency as:

O(n(n-1)) = O(n(n)) = O(n2)

The diagram analyzes the pseudocode implementation of bubble sort to show how we draw this conclusion.

When calculating the run-time efficiency of an algorithm, we drop the constant (-1), which simplifies our inner loop comparisons to n.

This is how we arrive at the algorithm’s runtime: O(n^2).

**Bubble Sort Review**

Bubble sort is an algorithm to sort a list through repeated swaps of adjacent elements. It has a runtime of O(n^2).

For nearly sorted lists, bubble sort performs relatively few operations since it only performs a swap when elements are out of order.

Bubble sort is a good introductory algorithm which opens the door to learning more complex algorithms. It answers the question, “How can we algorithmically sort a list?” and encourages us to ask, “How can we improve this sorting algorithm?”

BUBBLE SORT

bubble sort works by comparing a pair of neighboring elements and swapping their positions in the array so that the larger of the two elements is always on the right. Doing this continuously results in the largest element “bubbling” up to the end of the array, giving this sort its name. The algorithm only stops when there are no more values that need to be swapped.

Below is a quick pseudocode example of what we will create in this lesson:

while array is not sorted

for each value in array

if current value > next value

swap current value and next value

return array

Bubble sort is not the most efficient sorting algorithm. Bubble sort’s worst-case runtime is O(n^2). This is because we have to compare the current element we are looking at, to each element in the array after it and repeat this check for every single element in the array. Its best-case runtime is O(n) for an already-sorted list.

**BUBBLE SORT: JAVASCRIPT**

**Loops**

In order to sort an array, we’ll need to visit pairs of elements and check if they should be moved or kept at their current index. To accomplish this we’ll use two loops:

* One loop that will execute an inner loop depending on the state of a variable representing whether the input array might be sorted or not
* An inner loop to compare and swap pairs of elements in the array

### Instructions

**1.**

Begin by taking a look at the bubbleSort() function given to you in **bubbleSort.js**. Note that it takes one argument, an array to be sorted.

To start sorting, we will use a variable to store the condition of the input array as a Boolean value: true, meaning our input array might still be unsorted and need additional swaps of elements and we’ll later change it to false, meaning the input array does not need anymore swapping to sort it.

Add a variable inside bubbleSort() called swapping and assign it the value true.

Checkpoint 2 Passed

**2.**

Below the line where you declared swapping, create a while loop. This is the outer loop of our program that only runs if the input array might not be sorted and needs swapping, (the condition stored in swapping).

Use swapping which is currently set to true as the while condition.

This ensures that we’ll start running the while loop and run it at least once, since we need to loop through the input array at least one time to determine if it’s already sorted or needs swapping.

Checkpoint 3 Passed

**3.**

If we find that we don’t need to swap any of the elements, it means that the array is already sorted from smallest to largest and we can stop running our code and return the sorted array. To stop our while loop we only need to change the while condition to false.

Inside of the while loop we created, set swapping to false.

(We’ll add code later that will restart the loop if we might have to keep swapping to “bubble up” elements to the end of the array.)

Checkpoint 4 Passed

**4.**

Create a for loop nested inside the while loop under the line where you reassigned the value of swapping.

The for loop should visit every element in the input array starting from the first element and stopping at the second-to-last element. Setting the condition for the loop this way allows us to stay within the bounds of our input array and only check elements that exist.

Since the index is going to change, make sure to make it a let variable.

Checkpoint 5 Passed

**5.**

Lastly, bubbleSort() should return a sorted input array, (we’ll do the actual sorting in a later exercise).

Add code to return the sorted input array if we’ve exited our while loop.

Checkpoint 6 Passed

ANSWER: bubbleSort.js

const swap = require('./swap');

const bubbleSort = input => {

  let swapping = true;

  while (swapping) {

    swapping = false;

    for (let i = 0; i < input.length - 1; i++) {

    }

  }

  return input;

};

module.exports = bubbleSort;

swap.js

const swap = (arr, indexOne, indexTwo) => {

}

module.exports = swap;

**Swap**

An essential part of bubble sort is the “swapping” of pairs of unsorted elements. This animation illustrates how the swap() function of the bubble sort algorithm should work

To swap pairs of elements, we’ll need to store one of the values at either index in a temporary variable so we can use it later. For example, doing something like this:

currentValue = nextValue;  
nextValue = currentValue;

would not work because we’d “lose” one of the values. The original value of currentValue would be overwritten and there would be no reference to it. Using the temporary variable strategy seen in the GIF above avoids the loss of any of the values whose position we need to exchange.

We’ll employ this strategy to finish building out swap().

### Instructions

**1.**

Take a moment to look at the helper function in **swap.js**. Notice that swap() takes 3 arguments: the input array, the index of the current element, and finally, the index of the next element in the input array.

Create a constant called temp and store the value of the element at the indexTwo position in the input array so it can be referenced later.

Checkpoint 2 Passed

**2.**

Change the element at indexTwo of the input array to the value of the element at indexOne.

Checkpoint 3 Passed

**3.**

Change the element at indexOne of the input array to the original value of the element at indexTwo.

Checkpoint 4 Passed

Answer: swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

};

module.exports = swap;

bubbleSort.js

const swap = require('./swap');

const bubbleSort = input => {

  let swapping = true;

  while (swapping) {

    swapping = false;

    for (let i = 0; i < input.length - 1; i++) {

    }

  }

  return input;

};

module.exports = bubbleSort;

**Compare**

We have a loop to run our algorithm, and another that visits each element in the input array but if we were to run this as is, it would only iterate through the array once.

Let’s add some additional logic to our code in the **bubbleSort.js** file that will compare neighboring elements and swap them if necessary. For this exercise, you’ll only be adding code inside of the for loop of bubbleSort().

### Instructions

**1.**

Inside of the for loop we added, create a conditional that checks if the element at the current index is greater than the element one index after it.

Checkpoint 2 Passed

**2.**

Let’s begin adding the logic for the swapping action. To see how we’re changing our input array by swapping elements, add the following inside of the if statement you created:

console.log(`Swapping pair ${input[i]}, ${input[i+1]} in [${input}]`);

This code will log a message for every swap made when we execute bubbleSort().

Checkpoint 3 Passed

**3.**

Swap unordered pairs. After our logging statement, add a call to swap(), the helper function that handles changing the position of pairs of elements. Take a look at the parameters of swap() in the **swap.js** file to see what arguments you need to call it with.

Checkpoint 4 Passed

**4.**

If we make a swap, we want to loop through the array again to see if we need to make additional swaps to continue “bubbling up” elements in the wrong position.

Keep our while loop running by changing the value of the while condition variable so that it evaluates as true.

Checkpoint 5 Passed

Answer: bubbleSort.js

const swap = require('./swap');

const bubbleSort = input => {

  let swapping = true;

  while (swapping) {

    swapping = false;

    for (let i = 0; i < input.length - 1; i++) {

      if (input[i] > input[i + 1]) {

        console.log(`Swapping pair ${input[i]}, ${input[i+1]} in [${input}]`);

        swap(input, i, i + 1);

        swapping = true;

      }

    }

  }

  return input;

};

module.exports = bubbleSort;

console.log(bubbleSort([3, 2, 1]));

swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

};

module.exports = swap;

**Review**

**1.**

Test your work by calling bubbleSort() with each of the arrays below and logging the results:

[9, 8, 7, 6, 5, 4, 3, 2, 1]   
[1, 2, 3, 4, 5, 6, 7, 8, 9]

Notice how your implementation of bubble sort handles swapping elements in these two arrays. Do the number of swaps differ between the two arrays? Why would that be?

Checkpoint 2 Passed

**2.**

What real-life examples of bubble sort can you come up with?

Checkpoint 3 Passed

**3.**

Can you think of an input array that would result in an O(n^2) runtime? What input array would result in an O(n) runtime?

Checkpoint

Answer: bubbleSort.js

const swap = require('./swap');

const bubbleSort = input => {

  let swapCount = 0

  let swapping = true;

  while (swapping) {

    swapping = false;

    for (let i = 0; i < input.length - 1; i++) {

      if (input[i] > input[i + 1]) {

        swap(input, i, i + 1)

        swapCount++;

        swapping = true;

      }

    }

  }

  console.log(`Swapped ${swapCount} times`);

  return input;

};

bubbleSort([9, 8, 7, 6, 5, 4, 3, 2, 1]);

bubbleSort([1, 2, 3, 4, 5, 6, 7, 8, 9]);

module.exports = bubbleSort;

swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

};

module.exports = swap;