**Basics**

**Functions**

Console.log(); Basically prints to system.

You can create functions in two different methods.

function name() { }

let name = () => {}

let name = param => condition; also works in one line when you only have one param.

Parameters are attributes of functions that will use its values within the function. Located within ()

let myName = “Bruh Moment”

let name = (myName) => {

console.log(myName);

}

name(); This works because myName is global and can be used from every function.

name(myName); Also works;

can also set a default value

let name = (myName = “Guest”) => {} and will run “Guest” if myName is undefined;

**Return**

By default, even if a function runs correctly, it needs a return value so it will take that as the result of running the function

let myName = “Bruh Moment”

let name = (myName) => {

return myName ? true : false;

}

This will tell the system that the function will give the value of true;

let name = myName => myName ? true: false; also works

**Methods**

.length() takes the value of the string’s length and return a number.

.toUpperCase() changes string to all uppercase letters

.startsWith() checks the value of the first letter and the inputted value to check true or false

.trim() removes whitespace.

[**Math Methods**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

Math.random() takes a random value between 0 to 1. Multiply for bigger pool of numbers

Math.floor() rounds the number down to the nearest whole number.

Math.ceil() rounds the number up to the nearest whole number.

**Data Types**

Number: Any numeric value including those with decimals

String: A group of characters surrounded by quotes

Boolean: Data type that has only two values, true or false\

Null: Data type that has a value of null aka absence of a value.

Undefined: Data type that is similar to Null.

Symbol: Unique identifiers

Object: Collections of related data.

Variables: Stored data we created and define them with unique names

const is the static variable when you just want to keep it the same throughout the system.

let is the dynamic variable when you want to change the value through the system.

Don’t use var.

When we want to add a variable to a string to put on console.log, we can either use + or ${} using ``

console.log(“Bruh” + bruh) or console.log(`Bruh ${bruh}.`);

[**Arrays**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

Arrays consists of a list of elements in a single variable. Needs to include [] and separate items with ,

let array = [“One”, “Two”, “Three”];

When referencing an element of an array, start with 0 and then add one for each proceeding element.

let array = [“One”, “Two”, “Three”]; Then it means array[2] = Three;

Nested arrays are basically arrays that can be nested.

let array[1, [2, 3]];

array[1] = [2, 3] array[1][1] = 3

**Ops**

.join() will take every element of an array and form a single string statement

let array = [“One”, “Two”, “Three”];

array.join(“ ”);

Will result array reading as “One Two Three”

.push() allows us to add values to existing arrays

let array = [“One”, “Two”, “Three”]; array.push(“Four”);

Results in array = [“One”, “Two”, “Three”, “Four”];

.pop() is the opposite of .push where it would take out the right most element out of the array.

array.pop() will result in array to remove “Four”.

.shift() is similar to .pop, except it takes out the left most element out of the array.

.unshift() is the opposite of .shift, where it would add an element to the left side of the array.

.slice() can take up to two values to shorten the array.

let array = [“One”, “Two”, “Three”]; array.push(“Four”);

array.slice(0, 2) will return an array = [“One”, “Two”];

.splice() takes 3 values and it is used to replace elements in the array in a group

let array = [“One”, “Two”, “Three”];

array.splice(0, 3, “Four”) will result in starting at index 0, removing 3 elements, and replaced

array = [“Four”];

.indexOf() finds the index number of an element in an array

array.indexOf(“Three”) will equal 2.

**Objects**

Objects exist where they are written as such,

let bruh = {

name: “Bruh”,

occupation: “Moment”

};

Where typing bruh.name will return “Bruh”; bruh[“name”] also works.

delete is a command that will basically delete the property of the object

delete bruh.name will remove “Bruh” as a prop.

You can also create functions within the object by stating the name and adding () {}

let bruh = {

name: “Bruh”,

occupation: “Moment”,

sayBruh() {

console.log(“I said bruh”)

},

};

and running bruh.sayBruh() should work.

Nested objects also exist

let bruh = {

name: “Bruh”,

occupation: “Moment”,

sayBruh () {

console.log(“I said bruh”)

},

yeet: {

“yeet”: “Yeet”

}

};

bruh[yeet].yeet should return “Yeet”

**Pass by Reference**

Essentially you can run functions to change the values of Objects

let bruh = {

name: “Bruh”,

occupation: “Moment”,

sayBruh () {

console.log(“I said bruh”)

},

yeet: {

“yeet”: “Yeet”

}

};

const changeBruh = (obj) => {

obj.name = “Dab”,

obj.dab = “Dab More”

}

changeBruh(bruh) will result bruh.name = “Dab” and add a prop called dab.

**For loop using objects**

Objects can be used in stuff like for loops, when it iterates through the nested objects.

let vocabulary = {

bruh: {

definition: “To say something is bruh.”

},

dab: {

definition: “A form of motion to dab on them”

},

yeet: {

definition: “A battlecry revolving around yeet”

}

}

for (let i in vocabulary) {

console.log(vocabulary[i].definition)

}

This will log the 3 definitions of the vocabulary list.

**This**

It is a self-referencing keyword when using an object’s function to reference the object

let bruh = {

name: “Bruh”,

occupation: “Moment”,

sayBruh () {

console.log(“I said bruh”)

},

yeet: {

“yeet”: “Yeet”

}

changeBruh() {

this.name = “Dab”,

}

};

running bruh.changeBruh() will change bruh.name

**Get and return**

get is used to retrieve certain values of the object and return a signal that indicates if the information is there or not

let bruh = {

name: “Bruh”,

occupation: “Moment”,

sayBruh () {

console.log(“I said bruh”)

},

yeet: {

“yeet”: “Yeet”

}

get changeBruh() {

if (this.name && this.occupation) {

return true;

}

else return false;

}

};

**Set**

Set is used when you want to reset a value into an exist value

let user = {

username: “default”,

password: “123456789”,

set nameChange(newName) {

if (typeof newAge === “string”) {

this.username = newname;

}

else console.log(“Need letters.”)

},

}

**Using functions to create obj props**

Further more, we can create properties of an object with a function

const account = (username, password, zipcode, country) => {

return {

username,

password,

zipcode,

country

}

}

user.account(bruh, moment, 99999, USA) would create an account with those info.

**Destructured Assignment**

Instead of doing stuff like

const username = account.username

You can easily extract the info doing

const {username, password, zipcode, country} = account. To get more info, faster.

console.log(`username: ${username} password: ${password}`)

**Object Methods**

Objects.keys(obj) basically returns an array containing all the props of the obj

Objects.entries(obj) basically returns a nested array containing all the props and their values of the obj

Object.assign({prop: statement}, obj) Basically adds a prop with a value into the obj.

**Classes**

Classes are used to help simplify the amount of workload when it comes to dealing with objects and the functions associated with said objects.

class Bruh {

constructor(phrase, definition) {

this.\_name = phrase;

this.\_definition = definition;

}

get name() {

return this.\_name;

}

get definition() {

return this.\_definition;

}

set name(name) {

return this.\_name = name;

}

etc..

}

variables that are created with classes are called **instances**

const bruh = new Bruh(“bruh”, “This is a bruh moment.”);

Inheritance are basically parents of classes that share the same characteristics.

When the child wants to reference the parent via constructor, use super and then list the prop.

class Sayings {

constructor(phrase, definition) {

this.\_name = phrase;

this.\_definition = definition;

}

get name() {

return this.\_name;

}

get definition() {

return this.\_definition;

}

}

class Bruh extends Sayings {

constructor(phrase, definition, level) {

super(phrase, definition);

this.\_level = level;

}

}

static methods are methods that can only be called from the parent and child classes can not call it.

You basically just write static before the method

**Module**

Modules basically allow us to export pieces of code from one program to another

module.exports is the method that is used to export the code we want.

.require() would be how we import JS files in a different JS file.

Assuming we have a Bruh object

module.exports = Bruh;

On a different file you would do

const Bruh = require(“./bruh.js”);

export default thing basically is what we want to define as a default when importing to another file

When having multiple exports or imports, use {} to define what is being exported/imported

You can also just export functions that are being made, rather than exporting them at the bottom.

Sometimes when you export, function names are too long, so you would use “as” to shorten them

export {bruhmoment as bruh, yaaayeet as yeet};

Same thing can come from imports by using the \* symbol

import \* as Bruh from “./bruh” can allow you to do Bruh.bruh, Bruh.yeet;

**Errors**

throw Error() basically throws an error that we manually make, for instance if a user were to make a bad password then we can throw an error that indicates this.

try and catch. basically will try to predict an error and catch the error, to return a statement in case the error was caught.

try {

throw Error('This error will get caught');

} catch (e) {

console.log(e);

}

**Arithmetic operators**

Basic arithmetic includes (+, -, \*, /, and %)

% represents remainder of two numbers dividing.

You can also use = as a += such as w +=1 aka w = w + 1;

When using counters you can use ++ or -- to increment or decrement a value

w++ aka w+=1

**Other operators**

typeof checks inputted value’s data type.

**Conditional statements**

if () is a statement that asks if the condition is occurs. If the condition does happen, then an action will take place.

if (1 < 2) {

console.log(“True”);

}

if else () is a statement that is asked if the if statement is not true.

else is the final conditional statement when the if statement and every other if else statements are false.

Essentially runs when the whole procedure is false.

**“False” terms**

Whenever a value is being checked for either true or false, it will not be false unless it is equal to 0, an empty string, equal to null, equal to undefined, or equal to NaN.

0, “”, null, undefined, NaN

**Comparison Operators**

Helps checks the relation between two objects in a condition.

<, >, <=, >=, ===, and !== are the basic ones.

=== means that it is equal to while !== means it is not equal to

1 === 2 will run false and 1 !== 2 will run true

**Logical Operators**

logical operators are used to check two separate statements in a single condition

&& aka AND checks if both statements are true

|| aka OR checks if either one of the statements are true

! aka NOT will reverse if the statement is true/false

(1 === 1 && 1 > 2) will return false

(1 === 1 || 1 > 2) will return true

(!1 === 1) will return false

**Creating “Default” values**

Assume we are creating finding a username from our database, if the username doesn’t exist, then we treat the person as a “Guest”

This can be achieved by using the || such as the following.

let username = null;

let name = username || “Guest”

In this instance since username is null, it will make name be defined as “Guest”

**Ternary Operator**

This exists when you want to write an if, else statement in one line.

By utilizing ? and : we are able to achieve this in one line.

if (1 > 2) {

console.log(“Impossible”);

}

else {

console.log(“Naw duh”);

}

(1 > 2) ? console.log(“Impossible”) : console.log(“Naw duh”);

**Switch**

switch() is used as an alternative to using if else statements for easier syntax to read.

You would define the condition and then state what happens when a case happens. If all fails, it will be defined in the default section.

let number = 4;

switch (number) {

case 1:

console.log(“This is 1”);

break;

case 2:

console.log(“This is 2”);

break;

case 3:

console.log(“This is 3”);

break;

default:

console.log(“This is an unknown number”);

break;

}

**Loops**

break; will cause a loop to stop.

**For Loops**

for loops are loops that need 3 expressions to function. An initializer, a condition, and a counter.

let bruh = [1, 2, 3, 4]

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

console.log(bruh[i]);

}

Basically puts the value of 1 – 4 in the console on separate lines.

Nested for loop is basically a for loop within a for loop.

let bruh = [1, 2, 3, 4]

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

for (let j = 0; j < bruh.length; j++) {

console.log(bruh[i] + bruh[j]);

}

}

**While Loops**

While loop is basically, if this condition is still valid, then I will keep running.

let bruh = 0;

while (bruh < 5) {

bruh++;

} //Runs until bruh is equal to 5;

do while is the very similar to a regular while loop, except it will execute the code within do, before checking the while condition.

let bruh = 5;

do {

bruh++;

}

while (bruh < 5); // bruh will = 6 since it executes the do regardless of while condition.

**Iterators**

**For each Loops**

for each loops are different to where you would execute the same code for every element in an array.

It can be written in two ways;

const array = [1, 2, 3, 4];

array.forEach(function(numbers) {

console.log(numbers);

});

array.forEach(numbers => console.log(numbers));

**.map()**

.map it basically takes an array, executes the code, and then returns a new array with the updated information.

const array = [1, 2, 3, 4];

const arrayMax = array.map(numbers => {return numbers \* 10});

// arrayMax = [10, 20, 30, 40];

**.filter()**

Basically map, except it is used to check conditionals and create a new array through filtering.

const array = [1, 10, 100, 1000]

const bigNumbers = array.filter(numbers => {

return numbers > 10;

})

// bigNumbers = [100, 1000]’

**.findIndex()**

Basically will return the index of the first element that matches a condition

const array = [1, 2, 3, 4]

const two = array.findIndex(number => {

return (number > 1 && number < 3);

})

// two = [2];

**.reduce()**

Basically takes an array of data and turn it into a single element. It takes two params and take a second argument to serve as a “initial value”

const array= [1, 2, 3, 4]

const added= array.reduce((total, number) => {

return (number + total);

}, 1000)

// basically returns as 1 + 2 + 3 + 4 + 1000 = 1010

.some()

Basically checks the if the condition of some element in the array is true

.every() checks the if the condition of every element in the array is true

**Errors**

SyntaxError: Basically typo in code

ReferenceError: Would basically refer to saying the used variable doesn’t exist or is not referenced properly.

TypeError: The use of the variable is not used correctly. You cannot use a string variable in a method that only takes number params.

**DOM stuff**

When referencing the document and want to change the html/css within the js file, you would use document.

document.body.innerHTML would change the content of the body of HTML.

document.querySelector(“p”).innerHTML would change the content of the 1st paragraph element.

document.getElementById(“bruh”).innerHTML would change the content of the elements with the id listed as “bruh”.

.style basically accesses the css portion of the element

let bruh = document.getElementById(“bruh”);

bruh.style.backgroundcolor = “blue”;

You can even add stuff to the html parent by creating and inserting an element.

let bruh = document.createElement(“p”);

paragraph.id = “bruh”;

paragraph.innerHTML = “This is a bruh moment”

document.body.appendChild(bruh); //Will be the appended as last child of parent.

removeChild() is basically the opposite of appendChild()

**Effects**

onclick

document.getElementById(“Bruh”).onclick = function() {element.style.background = “black”};

When clicked, the background of bruh is black.

firstChild refers to the firstChild of the referenced parent

parentNode refers to the parent of the referenced child.

**Event Handler Registration**

.addEventListener() is used as a way for a specific DOM element to listen to a specific event.

let bruh = document.getElementById(“bruhTrigger”)

bruh.addEventListener(“click”, function)

//When this certain element is clicked, then the function will run

.removeEventListener() is used to reverse the addEventListener

**Event objects**

Event objects are used to gather data of user’s interaction with the website or application.

.target references the element that the event is registered to.

.type will access the name of the event

.timeStamp will return the amount of milliseconds that passed since the doc was loaded until the event was triggered.

**Mouse events**

onclick basically clicking will cause the event

wheel basically using the wheel of the mouse will cause the event

mousedown basically when you are clicking and holding the click

mouseup basically the moment when you release the click

mouseover basically when the mouse is over the element

mouseout basically when the mouse is went from inside to outside the element

**Asynchronous ops**

Asynchronous means doing something separately while doing another thing. Asynch objects will load by themselves, regardless of the status of the website loading, therefore it is more efficient.

Promise is described as the status of asynch object

pending means that the asynch request is being evaluated

fulfilled means if all the requirements are there during pending, it will run

rejected means if not all the requirements are there during pending, it will leave an error.

const executorFunction = (resolve, reject) => {

if (bruh = 1) resolve(“Bruh is right”);

else reject(“Bruh is not right”)

}

const myFirstBruh = new Promise(executorFunction);

setTimeOut() is an asynch function that takes a function and a delay timer to run.

const bruh = () => {console.log(“bruh”)};

setTimeOut(bruh, 1000); //prints bruh to console after 1 second.

Can also be used within a executorFunction

const executorFunctionTimer = () => {

return new Promise((r­esolve, reject) => {

setTimeOut(() => {resolve(“Bruh”)}, 1000)’

});

}­

Fulfilled and rejected functions. Basically, when something good or bad happens then you would want to think about what happens after.

…

const executorFunction = (resolve, reject) => {

if (bruh = 1) resolve(“Bruh is right”);

else reject(“Bruh is not right”)

}

const myFirstBruh = new Promise(executorFunction);

myFirstBruh.then(bruh, dab) //bruh will run if success, rejected would run dab;

Another way to write something like this would be a .then then a .catch statement;

myFirstBruh()

.then(bruh);

.catch(dab);

You can also chain multiple processes such as this

Assume there is BruhOne and BruhTwo where Bruh2 is a function that takes an accepted value from Bruh1

BruhOne()

.then((Bruh) => {

return BruhTwo(Bruh); //Creates BruhT

})

.then((BruhT) => {

console.log(BruhT)

})

You can also do multiple Promises at the same time using Promise.all

let myPromises = Promise.all([One(), Two(), Three()]);

myPromises

.then((Thing) => {

console.log(“Good”);

})

.catch((Rejection) => {

console.log(“Bad”);

})

**async** props

const Bruh = async () => {

console.log(“Bruh”);

};

Bruh();

.then(resolved => {

console.log(resolved)

});

await is basically stating we need to halt our execution so something could happen;

const asyncBruh = async () {

let resolved = await Bruh();

console.log(resolved);

}

From an earlier example about multiple async functions, we can simplify using this structure

const asyncBruh = async () => {

let resolved = await Bruh();

console.log(resolved);

let resolvedAgain = await BruhTwo(resolved);

console.log(resolvedAgain);

}

const failBruh = asynchBruh()

failBruh.catch((failure) => {

console.log(failure);

});

const asyncBruhAll = asynch () => {

const bruhAll = await Promiseall([one(), two(), three()]);

for (let i = 0, I < bruhAll.length, i++) {

console.log(bruhAll[i]);

}

}

**Nodes**

Nodes and trees are considered important concepts when it comes to describing data management.

A way to write Nodes is as a class with a constructor

class Node {

constructor(data) {

this.data = data;

}

}

const firstNode = new Node(“Bruh”);

console.log(firstNode.data) // returns Bruh

**Codecademy examples**

.sort()

const speciesArray = [ {speciesName:'shark', numTeeth:50}, {speciesName:'dog', numTeeth:42}, {speciesName:'alligator', numTeeth:80}, {speciesName:'human', numTeeth:32}];

const sortSpeciesByTeeth = arr => arr.sort((speciesObj1, speciesObj2) => speciesObj1.numTeeth > speciesObj2.numTeeth)

**Nodes**

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;

}

this.head = removedHead.getNextNode();

return removedHead.data;

}

printList() {

let currentNode = this.head;

let output = '<head> ';

while (currentNode !== null) {

output += currentNode.data + ' ';

currentNode = currentNode.getNextNode();

}

output += '<tail>';

console.log(output);

}

}

module.exports = LinkedList;

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

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

}

addToHead(data) {

const newHead = new Node(data);

const currentHead = this.head;

if (currentHead) {

currentHead.setPreviousNode(newHead);

newHead.setNextNode(currentHead);

}

this.head = newHead;

if (!this.tail) {

this.tail = newHead;

}

}

addToTail(data) {

const newTail = new Node(data);

const currentTail = this.tail;

if (currentTail) {

currentTail.setNextNode(newTail);

newTail.setPreviousNode(currentTail);

}

this.tail = newTail;

if (!this.head) {

this.head = newTail;

}

}

removeHead() {

const removedHead = this.head;

if (!removedHead) {

return;

}

this.head = removedHead.getNextNode();

if (this.head) {

this.head.setPreviousNode(null);

}

if (removedHead === this.tail) {

this.removeTail();

}

return removedHead.data;

}

removeTail() {

const removedTail = this.tail;

if (!removedTail) {

return;

}

this.tail = removedTail.getPreviousNode();

if (this.tail) {

this.tail.setNextNode(null);

}

if (removedTail === this.head) {

this.removeHead();

}

return removedTail.data;

}

removeByData(data) {

let nodeToRemove;

let currentNode = this.head;

while (currentNode !== null) {

if (currentNode.data === data) {

nodeToRemove = currentNode;

break;

}

currentNode = currentNode.getNextNode();

}

if (!nodeToRemove) {

return null;

}

if (nodeToRemove === this.head) {

this.removeHead();

} else if (nodeToRemove === this.tail) {

this.removeTail();

} else {

const nextNode = nodeToRemove.getNextNode();

const previousNode = nodeToRemove.getPreviousNode();

nextNode.setPreviousNode(previousNode);

previousNode.setNextNode(nextNode);

}

return nodeToRemove;

}

printList() {

let currentNode = this.head;

let output = '<head> ';

while (currentNode !== null) {

output += currentNode.data + ' ';

currentNode = currentNode.getNextNode();

}

output += '<tail>';

console.log(output);

}

}

module.exports = DoublyLinkedList;

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

class Queue {

constructor(maxSize = Infinity) {

this.queue = new LinkedList();

this.maxSize = maxSize;

this.size = 0;

}

isEmpty() {

return this.size === 0;

}

hasRoom() {

return this.size < this.maxSize;

}

enqueue(data) {

if (this.hasRoom()) {

this.queue.addToTail(data);

this.size++;

} else {

throw new Error("Queue is full!");

}

}

dequeue() {

if (!this.isEmpty()) {

const data = this.queue.removeHead();

this.size--;

return data;

} else {

throw new Error("Queue is empty!");

}

}

}

module.exports = Queue;

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

class Stack {

constructor(maxSize = Infinity) {

this.stack = new LinkedList();

this.maxSize = maxSize;

this.size = 0;

}

// Add helper methods below this line

hasRoom() {

return this.size < this.maxSize;

}

isEmpty() {

return this.size === 0;

}

push(value) {

if (this.hasRoom()) {

this.stack.addToHead(value);

this.size++;

} else {

throw new Error('Stack is full');

}

}

pop() {

if (!this.isEmpty()) {

const value = this.stack.removeHead();

this.size--;

return value;

} else {

throw new Error('Stack is empty');

}

}

peek() {

if (!this.isEmpty()) {

return this.stack.head.data;

} else {

return null;

}

}

}

module.exports = Stack;

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;

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;

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;

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;

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;

class Edge {

constructor(start, end, weight = null) {

this.start = start;

this.end = end;

this.weight = weight;

}

}

module.exports = Edge;

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;

}

const swap = (arr, indexOne, indexTwo) => {

const temp = arr[indexTwo];

arr[indexTwo] = arr[indexOne];

arr[indexOne] = temp;

};

module.exports = swap;

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;

};

module.exports = bubbleSort;

const mergeSort = (startArray) => {

const length = startArray.length;

if (length === 1) {

return startArray;

}

const mid = Math.floor(length / 2);

const leftArray = startArray.slice(0, mid);

const rightArray = startArray.slice(mid, length);

return merge(mergeSort(leftArray), mergeSort(rightArray))

}

const merge = (leftArray, rightArray) => {

const sortedArray = [];

while (leftArray.length > 0 && rightArray.length > 0) {

if (leftArray[0] < rightArray[0]) {

sortedArray.push(leftArray[0]);

leftArray.shift();

} else {

sortedArray.push(rightArray[0]);

rightArray.shift();

}

}

return sortedArray.concat(leftArray).concat(rightArray);

}

const inputArr = [3, 5, 2, 90, 4, 7];

console.log(mergeSort(inputArr));

module.exports = {

mergeSort

};

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

const quicksort = (array, leftBound = 0, rightBound = array.length - 1) => {

if (leftBound < rightBound) {

const pivotIndex = partition(array, leftBound, rightBound);

quicksort(array, leftBound, pivotIndex - 1);

quicksort(array, pivotIndex, rightBound);

}

return array;

}

const partition = (array, leftIndex, rightIndex) => {

const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

while (leftIndex <= rightIndex) {

while (array[leftIndex] < pivot) {

leftIndex++;

}

while (array[rightIndex] > pivot) {

rightIndex--;

}

if (leftIndex <= rightIndex) {

swap(array, leftIndex, rightIndex);

leftIndex++;

rightIndex--;

}

}

return leftIndex;

}

module.exports = {

quicksort,

partition

};

const binarySearch = (arr, target) => {

let left = 0;

let right = arr.length;

while (right > left) {

const indexToCheck = Math.floor((left + right) / 2);

const checking = arr[indexToCheck];

console.log(indexToCheck);

if (checking === target) {

return indexToCheck;

} else if (checking < target) {

left = indexToCheck + 1;

} else {

right = indexToCheck;

}

}

return null;

}

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];

const target = 41;

targetIndex = binarySearch(searchable, target);

console.log(`The target index is ${targetIndex}.`);

module.exports = binarySearch;

class BinaryTree {

constructor(value, depth = 1) {

this.value = value;

this.depth = depth;

this.left = null;

this.right = null;

}

insert(value) {

if (value < this.value) {

if (!this.left) {

this.left = new BinaryTree(value, this.depth + 1);

} else {

this.left.insert(value);

}

} else {

if (!this.right) {

this.right = new BinaryTree(value, this.depth + 1);

} else {

this.right.insert(value);

}

}

}

getNodeByValue(value) {

if (this.value === value) {

return this;

} else if ((this.left) && (value < this.value)) {

return this.left.getNodeByValue(value);

} else if (this.right) {

return this.right.getNodeByValue(value);

} else {

return null;

}

}

depthFirstTraversal() {

if (this.left) {

this.left.depthFirstTraversal();

}

console.log(`Depth=${this.depth}, Value=${this.value}`);

if (this.right) {

this.right.depthFirstTraversal();

}

}

};

module.exports = BinaryTree;

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph;

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

const breadthFirstTraversal = (start) => {

const visitedVertices = [start];

start.edges.forEach(edge => {

const neighbor = edge.end;

if (!visitedVertices.includes(neighbor)) {

visitedVertices.push(neighbor)

}

});

console.log(visitedVertices)

};

breadthFirstTraversal(testGraph.vertices[0]);

**~~React (Outdated)~~**

~~Whenever you use react, always import React from “react”;~~

~~Whenever using a DOM, always import ReactDOM from “react-dom”;~~

**~~ReactDOM~~**

~~To utilize react in a webpage it is important to identify which JSX expression you want to run as well as which html file you want to link with so it will render.)~~

~~ReactDOM.render(thing, document.getElementById(“app”));~~

~~//html file~~

~~<body>~~

~~<main id=“app”>~~

~~</main>~~

~~</body>~~

~~If this is ran, it would look similar to this~~

~~<main id= “app”>~~

~~Whatever thing is;~~

~~</main>~~

**~~JSX Elements~~**

~~Basically can write html tags within a js file. You can also create variables that associate with html tags~~

~~const bruh = <h1>Bruh</h1>;~~

~~if there are multiple lines of html code, you would need to enclose them with ()~~

~~const bruh = (~~

~~<div id=“bruh” className= “bruh”>~~

~~<p>bruh</p>~~

~~<h2>Bruh</h2>~~

~~<h1>BRUH</h1>~~

~~</div>~~

~~);~~

~~Note: Notice you can’t use class, you gotta use className;~~

~~Note: You cannot have more than one outermost element, therefore it is recommended to use a div and insert those elements within it.~~

~~To include JavaScript elements in JSX, we need to include {} so that way it is read as JS~~

~~const bruh = <h1> 2 + 3 </h1> is different from const bruh = <h1>{2 + 3}</h1>.~~

~~One returns 2 + 3 and other returns 5;\~~

**~~Variable attributes~~**

~~An effective way to use variables are creating attributes that can be referenced later on~~

~~const picture = {~~

~~length: “50%”,~~

~~bruh: “../images/bruh.png”;~~

~~yeet: “../images/yeet.png”;~~

~~}~~

~~const warning = () => {alert(“Bruh, this is a warning.”)}~~

~~const bruh = (~~

~~<img src={picture.bruh} alt=”Bruh” height={picture.length onClick={warning}} />~~

~~<h1>Picture of a bruh. Don’t click pls</h1>~~

~~)~~

**~~Conditional Statements~~**

~~When using JSX, it cannot compile conditional statements, such as if statements. One way to handle it is call these conditional statements outside the JSX. Example from Codecademy~~

~~function coinToss() {~~

~~// This function will randomly return either 'heads' or 'tails'.~~

~~return Math.random() < 0.5 ? 'heads' : 'tails';~~

~~}~~

~~const pics = {~~

~~kitty: 'https://content.codecademy.com/courses/React/react\_photo-kitty.jpg',~~

~~doggy: 'https://content.codecademy.com/courses/React/react\_photo-puppy.jpeg'~~

~~};~~

~~let img;~~

~~// if/else statement begins here:~~

~~if (coinToss() === "heads") {img = <img src={pics.kitty} />}~~

~~else {img = <img src={pics.doggy} />}~~

~~You can also just write the ternary op which is way easier and more effective~~

~~const pics = {~~

~~kitty: 'https://content.codecademy.com/courses/React/react\_photo-kitty.jpg',~~

~~doggy: 'https://content.codecademy.com/courses/React/react\_photo-puppy.jpeg'~~

~~};~~

~~let img;~~

~~let cat = <img src={pics.kitty} />~~

~~const coinToss() {~~

~~// returns img changing. If Math.random() < 0.5 then pics[kitty] else pics[doggy]~~

~~return img = <img src={pics[Math.random() < 0.5 ? "kitty" : "doggy"]}/>~~

~~}~~

**~~&&~~** ~~is not the same as the one used for other platforms. What it does it checks the first statement, if first statement is true, then it runs the second statement~~

~~let bruh = “bruh”;~~

~~const statement = (~~

~~{bruh === “bruh” && <h1>BRUH</h1>}~~

~~// will show BRUH because of bruh does = “bruh”~~

~~)~~

**~~maps~~**

~~Similar to how maps work in JS.~~

~~const phrases = [“Bruh”, “Dab”, “Yeet”];~~

~~const listPhrases = phrases.map(phrase => <li>{phrase}</li>);~~

~~const result = <ul>{listPhrases}</ul> //Will show unlist containing 3 list elements;~~

**~~keys~~**

~~Basically just think of them as IDs so.~~

~~<p key=”bruh”>Bruh</p>~~

~~Keys in map~~

~~const phrases = [“Bruh”, “Dab”, “Yeet”];~~

~~const listPhrases = phrases.map((phrase, i) => <li key={i}>{phrase}</li>);~~

~~const result = <ul>{listPhrases}</ul> //Will show unlist containing 3 list elements with keys representing 0, 1, and 2;~~

**~~createElement~~**

~~Basically you can write React code without JSX~~

~~const h1 = <h1>Bruh</h1> is also~~

~~const h1 = React.createElement(“h1”, null, “Bruh”);~~

~~Note: The order of sequence for the following~~

~~a. Changes on the real DOM cause the screen to change.~~

~~b. A JSX element renders.~~

~~c. The virtual DOM is compared to what it looked like before it updated to figure out which objects have changed.~~

~~d. The entire virtual DOM gets updated.~~

~~e. The changed objects, and the changed objects only, get updated on the real DOM.~~

~~Is b, d, c, e, a.~~

**~~React Components~~**

~~components are basically reusable chunk of code that involves rendering HTML~~

~~class Bruh extends React.Component {~~

~~}~~

~~note: Don’t camelCase it;~~

~~Afterwards you would create a render function that returns a jsx expression~~

~~class Bruh extends React.Component {~~

~~render() {~~

~~const bruh = “Bruh”;~~

~~get bbruh() {return “Bruh”}~~

~~return (~~

~~<div>~~

~~<h1>Bruh</h1>~~

~~<h1>{bruh}</h1>~~

~~<h1>{this.bbruh}</h1>~~

~~</div>~~

~~)~~

~~}~~

~~}~~

~~Afterwards you can make an instance by calling <Bruh />~~

**~~Props~~**

~~Props are basically properties that we can put on our components~~

~~<Bruh string= “Bruh” /> Where <Bruh> has a prop that is named string with “Bruh”~~

~~Therefore if we want to call it, we can use this.props.string to call it, or~~

~~you can also do this.props.children. Works only if you are referencing a single item.~~

~~This will work even if the function itself is on a different file as long as the prop is referenced locally.~~

~~You can also pass methods as props~~

~~class Bruh extends React.Component {~~

~~render() {~~

~~Talk() {~~

~~return console.log(“bruh”)~~

~~}~~

~~return (~~

~~<h1>Important Information</h1>~~

~~<button onClick= {this.talk}></button>~~

~~)~~

~~}~~

~~}~~

~~defaultProps is a property that can help create default props in case there is no information given.~~

~~class Bruh extends React.Component {~~

~~render() {~~

~~Talk() {~~

~~return console.log(“bruh”)~~

~~}~~

~~return (~~

~~<h1>Important Information</h1>~~

~~<button onClick= {this.talk}>{this.props.bruh}</button>~~

~~)~~

~~}~~

~~}~~

~~Bruh.defaultProps(bruh = “Button”)~~

~~Props into Information~~

~~Let’s assume we have a this as a prop~~

~~Bruh …~~

~~return (~~

~~<h1>Bruh</h1>~~

~~<h1>Moment</h1>~~

~~);~~

~~We can take this and call (this.props.children instanceof Array) What it will do is that it will check if this.props contain more than one children since it will count as an array with 2 or more items.~~