Stacks

In computer science, a **stack** is an abstract data type that serves as a collection of elements, with two principal operations:

* **push**, which adds an element to the collection, and
* **pop**, which removes the most recently added element that was not yet removed.

It is a LIFO data structure.

/\* Stacks! \*/

// functions: push, pop, peek, length

var letters = []; // this is our stack

var word = "freeCodeCamp"

var rword = "";

// put letters of word into stack

for (var i = 0; i < word.length; i++) {

letters.push(word[i]);

}

// pop off the stack in reverse order

for (var i = 0; i < word.length; i++) {

rword += letters.pop();

}

if (rword === word) {

console.log(word + " is a palindrome.");

}

else {

console.log(word + " is not a palindrome.");

}

// Creates a stack

var Stack = function() {

this.count = 0;

this.storage = {};

// Adds a value onto the end of the stack

this.push = function(value) {

this.storage[this.count] = value;

this.count++;

}

// Removes and returns the value at the end of the stack

this.pop = function() {

if (this.count === 0) {

return undefined;

}

this.count--;

var result = this.storage[this.count];

delete this.storage[this.count];

return result;

}

this.size = function() {

return this.count;

}

// Returns the value at the end of the stack

this.peek = function() {

return this.storage[this.count-1];

}

}

var myStack = new Stack();

myStack.push(1);

myStack.push(2);

console.log(myStack.peek());

console.log(myStack.pop());

console.log(myStack.peek());

myStack.push("freeCodeCamp");

console.log(myStack.size());

console.log(myStack.peek());

console.log(myStack.pop());

console.log(myStack.peek());

Sets

/\* Sets \*/

function mySet() {

// the var collection will hold the set

var collection = [];

// this method will check for the presence of an element and return true or false

this.has = function(element) {

return (collection.indexOf(element) !== -1);

};

// this method will return all the values in the set

this.values = function() {

return collection;

};

// this method will add an element to the set

this.add = function(element) {

if(!this.has(element)){

collection.push(element);

return true;

}

return false;

};

// this method will remove an element from a set

this.remove = function(element) {

if(this.has(element)){

index = collection.indexOf(element);

collection.splice(index,1);

return true;

}

return false;

};

// this method will return the size of the collection

this.size = function() {

return collection.length;

};

// this method will return the union of two sets

this.union = function(otherSet) {

var unionSet = new mySet();

var firstSet = this.values();

var secondSet = otherSet.values();

firstSet.forEach(function(e){

unionSet.add(e);

});

secondSet.forEach(function(e){

unionSet.add(e);

});

return unionSet;

};

// this method will return the intersection of two sets as a new set

this.intersection = function(otherSet) {

var intersectionSet = new mySet();

var firstSet = this.values();

firstSet.forEach(function(e){

if(otherSet.has(e)){

intersectionSet.add(e);

}

});

return intersectionSet;

};

// this method will return the difference of two sets as a new set

this.difference = function(otherSet) {

var differenceSet = new mySet();

var firstSet = this.values();

firstSet.forEach(function(e){

if(!otherSet.has(e)){

differenceSet.add(e);

}

});

return differenceSet;

};

// this method will test if the set is a subset of a different set

this.subset = function(otherSet) {

var firstSet = this.values();

return firstSet.every(function(value) {

return otherSet.has(value);

});

};

}

var setA = new mySet();

var setB = new mySet();

setA.add("a");

setB.add("b");

setB.add("c");

setB.add("a");

setB.add("d");

console.log(setA.subset(setB));

console.log(setA.intersection(setB).values());

console.log(setB.difference(setA).values());

var setC = new Set();

var setD = new Set();

setC.add("a");

setD.add("b");

setD.add("c");

setD.add("a");

setD.add("d");

console.log(setD.values())

setD.delete("a");

console.log(setD.has("a"));

console.log(setD.add("d"));

Queue

In computer science, a **queue** is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principle (or only) operations on the collection are the addition of entities to the rear terminal position, known as enqueue, and removal of entities from the front terminal position, known as dequeue. This makes the queue a First-In-First-Out (FIFO) data structure.

/\* Queues \*/

function Queue () {

collection = [];

this.print = function() {

console.log(collection);

};

this.enqueue = function(element) {

collection.push(element);

};

this.dequeue = function() {

return collection.shift();

};

this.front = function() {

return collection[0];

};

this.size = function() {

return collection.length;

};

this.isEmpty = function() {

return (collection.length === 0);

};

}

var q = new Queue();

q.enqueue('a');

q.enqueue('b');

q.enqueue('c');

q.print();

q.dequeue();

console.log(q.front());

q.print();

Priority Queue

Priority Queue is an extension of Queue having some properties as follows:

1. In Priority queue items are ordered by key (priority) value.
2. Item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.
3. Lower the value, higher the priority.

function PriorityQueue () {

var collection = [];

this.printCollection = function() {

(console.log(collection));

};

this.enqueue = function(element){

if (this.isEmpty()){

collection.push(element);

} else {

var added = false;

for (var i=0; i<collection.length; i++){

if (element[1] < collection[i][1]){ //checking priorities

collection.splice(i,0,element);

added = true;

break;

}

}

if (!added){

collection.push(element);

}

}

};

this.dequeue = function() {

var value = collection.shift();

return value[0];

};

this.front = function() {

return collection[0];

};

this.size = function() {

return collection.length;

};

this.isEmpty = function() {

return (collection.length === 0);

};

}

var pq = new PriorityQueue();

pq.enqueue(['Beau Carnes', 2]);

pq.enqueue(['Quincy Larson', 3]);

pq.enqueue(['Ewa Mitulska-Wójcik', 1])

pq.enqueue(['Briana Swift', 2])

pq.printCollection();

pq.dequeue();

console.log(pq.front());

pq.printCollection();