stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
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());
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
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"));
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
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();

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 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();
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
class Node {
constructor(data, left = null, right = null) {
this.data = data;
this.left = left;
this.right = right;
}
}

class BST {
constructor() {
this.root = null;
}

add(data) {
const node = this.root;
if (node === null) {
this.root = new Node(data);
return;
}
else {
const searchTree = function(node) {
if (data < node.data) {
if (node.left === null) {
node.left = new Node(data);
return;
} else if (node.left !== null) {
return searchTree(node.left);
}
} else if (data > node.data) {
if (node.right === null) {
node.right = new Node(data);
return;
} else if (node.right !== null) {
return searchTree(node.right);
}
}
else {
return null;
}
};
return searchTree(node);
}
}

findMin() {
let current = this.root;
while (current.left !== null) {
current = current.left;
}
return current.data;
}

findMax() {
let current = this.root;
while (current.right !== null) {
current = current.right;
}
return current.data;
}

find(data) {
let current = this.root;
while (current.data !== data) {
if (data < current.data) {
current = current.left;
}
else {
```

```
current = current.right;
}

if (current === null) {
return null;
}
}
return current;
}

isPresent(data) {
let current = this.root;
while (current) {
if (data === current.data) {
return true;
}

if (data < current.data) {
current = current.left;
} else {
current = current.right;
}
}
return false;
}

remove(data) {
const removeNode = function(node, data) {
if (node == null) {
return null;
}
if (data == node.data) {

// node has no children
if (node.left == null && node.right == null) {
return null;
}

// node has no left child
if (node.left == null) {
return node.right;
}

// node has no right child
if (node.right == null) {
return node.left;
}

// node has two children
var tempNode = node.right;
while (tempNode.left !== null) {
tempNode = tempNode.left;
}

node.data = tempNode.data;
node.right = removeNode(node.right, tempNode.data);
return node;
}
else if (data < node.data) {
node.left = removeNode(node.left, data);
return node;
}
else {
node.right = removeNode(node.right, data);
return node;
}
}
this.root = removeNode(this.root, data);
}
```

```
isBalanced() {
return (this.findMinHeight() >= this.findMaxHeight() - 1)
}

findMinHeight(node = this.root) {
if (node == null) {
return -1;
};

let left = this.findMinHeight(node.left);
let right = this.findMinHeight(node.right);
if (left < right) {
return left + 1;
}
else {
return right + 1;
};
}

findMaxHeight(node = this.root) {
if (node == null) {
return -1;
};

let left = this.findMaxHeight(node.left);
let right = this.findMaxHeight(node.right);
if (left > right) {
return left + 1;
}
else {
return right + 1;
};
}

inOrder() {
if (this.root == null) {
return null;
}
else {
var result = new Array();
function traverseInOrder(node) {
node.left && traverseInOrder(node.left);
result.push(node.data);
node.right && traverseInOrder(node.right);
}

traverseInOrder(this.root);
return result;
};
}

preOrder() {
if (this.root == null) {
return null;
} else {
var result = new Array();
function traversePreOrder(node) {
result.push(node.data);
node.left && traversePreOrder(node.left);
node.right && traversePreOrder(node.right);
};

traversePreOrder(this.root);
return result;
};
}

postOrder() {
if (this.root == null) {
return null;
```

```
} else {
var result = new Array();
function traversePostOrder(node) {
node.left && traversePostOrder(node.left);
node.right && traversePostOrder(node.right);
result.push(node.data);
};

traversePostOrder(this.root);
return result;
}
}

levelOrder() {
let result = [];
let Q = [];

if (this.root != null) {
Q.push(this.root);
while(Q.length > 0) {
let node = Q.shift();
result.push(node.data);
if (node.left != null) {
Q.push(node.left);
};

if (node.right != null) {
Q.push(node.right);
};
};
return result;
} else {
return null;
};
};
}

const bst = new BST();
bst.add(9);
bst.add(4);
bst.add(17);
bst.add(3);
bst.add(6);
bst.add(22);
bst.add(5);
bst.add(7);
bst.add(20);

console.log(bst.findMinHeight());
console.log(bst.findMaxHeight());
console.log(bst.isBalanced());
bst.add(10);
console.log(bst.findMinHeight());
console.log(bst.findMaxHeight());
console.log(bst.isBalanced());
console.log('inOrder: ' + bst.inOrder());
console.log('preOrder: ' + bst.preOrder());
console.log('postOrder: ' + bst.postOrder());

console.log('levelOrder: ' + bst.levelOrder());
```

stack
sets
queues
binaryST
hash
linkedList
trie

Stack*functions: push, pop, peek, length*

Home

```
var hash = (string, max) => {
var hash = 0;

for (var i = 0; i < string.length; i++) {
hash += string.charCodeAt(i);
```

heaps
graphs

```
}
return hash % max;
};

let HashTable = function() {

let storage = [];
const storageLimit = 14;

this.print = function() {
console.log(storage)
}

this.add = function(key, value) {
var index = hash(key, storageLimit);
if (storage[index] === undefined) {
storage[index] = [
[key, value]
];
}
else {
var inserted = false;
for (var i = 0; i < storage[index].length; i++) {
if (storage[index][i][0] === key) {
storage[index][i][1] = value;
inserted = true;
}
}
if (inserted === false) {
storage[index].push([key, value]);
}
}
};

this.remove = function(key) {
var index = hash(key, storageLimit);
if (storage[index].length === 1 && storage[index][0][0] === key) {
delete storage[index];
} else {
for (var i = 0; i < storage[index].length; i++) {
if (storage[index][i][0] === key) {
delete storage[index][i];
}
}
}
};

this.lookup = function(key) {
var index = hash(key, storageLimit);
if (storage[index] === undefined) {
return undefined;
} else {
for (var i = 0; i < storage[index].length; i++) {
if (storage[index][i][0] === key) {
return storage[index][i][1];
}
}
}
};
};

console.log(hash('quincy', 10))
let ht = new HashTable();
ht.add('beau', 'person');
ht.add('fido', 'dog');
ht.add('rex', 'dinosour');
ht.add('tux', 'penguin')
console.log(ht.lookup('tux'))
ht.print();
```

Stack*functions: push, pop, peek, length*

Home

```
function LinkedList() {
var length = 0;
var head = null;

var Node = function(element){
this.element = element;
this.next = null;
};

this.size = function(){
return length;
};

this.head = function(){
return head;
};

this.add = function(element){
var node = new Node(element);
if(head === null){
head = node;
} else {
var currentNode = head;

while(currentNode.next){
currentNode = currentNode.next;
}

currentNode.next = node;
}

length++;
};

this.remove = function(element){
var currentNode = head;
var previousNode;
if(currentNode.element === element){
head = currentNode.next;
} else {
while(currentNode.element !== element) {
previousNode = currentNode;
currentNode = currentNode.next;
}

previousNode.next = currentNode.next;
}

length --;
};

this.isEmpty = function() {
return length === 0;
};

this.indexOf = function(element) {
var currentNode = head;
var index = -1;

while(currentNode){
index++;
if(currentNode.element === element){
return index;
}
currentNode = currentNode.next;
}
```

```
return -1;
};

this.elementAt = function(index) {
var currentNode = head;
var count = 0;
while (count < index){
count ++;
currentNode = currentNode.next
}
return currentNode.element;
};

this.addAt = function(index, element){
var node = new Node(element);

var currentNode = head;
var previousNode;
var currentIndex = 0;

if(index > length){
return false;
}

if(index === 0){
node.next = currentNode;
head = node;
} else {
while(currentIndex < index){
currentIndex++;
previousNode = currentNode;
currentNode = currentNode.next;
}
node.next = currentNode;
previousNode.next = node;
}
length++;
}

this.removeAt = function(index) {
var currentNode = head;
var previousNode;
var currentIndex = 0;
if (index < 0 || index >= length){
return null
}

if(index === 0){
head = currentNode.next;
} else {
while(currentIndex < index) {
currentIndex ++;
previousNode = currentNode;
currentNode = currentNode.next;
}
previousNode.next = currentNode.next
}
length--;
return currentNode.element;
}
}
var conga = new LinkedList();
conga.add('Kitten');
conga.add('Puppy');
conga.add('Dog');
conga.add('Cat');
conga.add('Fish');
console.log(conga.size());
console.log(conga.removeAt(3));
console.log(conga.elementAt(3));
```

```
console.log(conga.indexOf('Puppy'));
console.log(conga.size());
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
let Node = function() {
this.keys = new Map();
this.end = false;
this.setEnd = function() {
this.end = true;
};
this.isEnd = function() {
return this.end;
};
};

let Trie = function() {
this.root = new Node();

this.add = function(input, node = this.root) {
if (input.length == 0) {
node.setEnd();
return;
} else if (!node.keys.has(input[0])) {
node.keys.set(input[0], new Node());
return this.add(input.substr(1), node.keys.get(input[0]));
} else {
return this.add(input.substr(1), node.keys.get(input[0]));
};
};

this.isWord = function(word) {
let node = this.root;
while (word.length > 1) {
if (!node.keys.has(word[0])) {
return false;
} else {
node = node.keys.get(word[0]);
word = word.substr(1);
};
};
return (node.keys.has(word) && node.keys.get(word).isEnd()) ? true : false;
};

this.print = function() {
let words = new Array();
let search = function(node, string) {
if (node.keys.size != 0) {
for (let letter of node.keys.keys()) {
search(node.keys.get(letter), string.concat(letter));
};
if (node.isEnd()) {
words.push(string);
};
} else {
string.length > 0 ? words.push(string) : undefined;
return;
};
};
search(this.root, new String());
return words.length > 0 ? words : mo;
};
};

myTrie = new Trie()
myTrie.add('ball');
myTrie.add('bat');
myTrie.add('doll');
```

```
myTrie.add('dork');
myTrie.add('do');
myTrie.add('dorm')
myTrie.add('send')
myTrie.add('sense')

console.log(myTrie.isWord('doll'))
console.log(myTrie.isWord('dor'))
console.log(myTrie.isWord('dorf'))
console.log(myTrie.print())
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
// left child: i * 2
// right child: i * 2 + 1
// parent: i / 2

let MinHeap = function() {

let heap = [null];

this.insert = function(num) {
heap.push(num);
if (heap.length > 2) {
let idx = heap.length - 1;
while (heap[idx] < heap[Math.floor(idx/2)]) {
if (idx >= 1) {
[heap[Math.floor(idx/2)], heap[idx]] = [heap[idx], heap[Math.floor(idx/2)]];
if (Math.floor(idx/2) > 1) {
idx = Math.floor(idx/2);
} else {
break;
};
};
};
};
};

this.remove = function() {
let smallest = heap[1];
if (heap.length > 2) {
heap[1] = heap[heap.length - 1];
heap.splice(heap.length - 1);
if (heap.length == 3) {
if (heap[1] > heap[2]) {
[heap[1], heap[2]] = [heap[2], heap[1]];
};
return smallest;
};
let i = 1;
let left = 2 * i;
let right = 2 * i + 1;
while (heap[i] >= heap[left] || heap[i] >= heap[right]) {
if (heap[left] < heap[right]) {
[heap[i], heap[left]] = [heap[left], heap[i]];
i = 2 * i
} else {
[heap[i], heap[right]] = [heap[right], heap[i]];
i = 2 * i + 1;
};
left = 2 * i;
right = 2 * i + 1;
if (heap[left] == undefined || heap[right] == undefined) {
break;
};
};
} else if (heap.length == 2) {
heap.splice(1, 1);
```

```
} else {
return null;
};
return smallest;
};

this.sort = function() {
let result = new Array();
while (heap.length > 1) {
result.push(this.remove());
};
return result;
};
};

let MaxHeap = function() {
let heap = [null];
this.print = () => heap;
this.insert = function(num) {
heap.push(num);
if (heap.length > 2) {
let idx = heap.length - 1;
while (heap[idx] > heap[Math.floor(idx/2)]) {
if (idx >= 1) {
[heap[Math.floor(idx/2)], heap[idx]] = [heap[idx], heap[Math.floor(idx/2)]];
if (Math.floor(idx/2) > 1) {
idx = Math.floor(idx/2);
} else {
break;
};
};
};
};
};

this.remove = function() {
let smallest = heap[1];
if (heap.length > 2) {
heap[1] = heap[heap.length - 1];
heap.splice(heap.length - 1);
if (heap.length == 3) {
if (heap[1] < heap[2]) {
[heap[1], heap[2]] = [heap[2], heap[1]];
};
return smallest;
};

let i = 1;
let left = 2 * i;
let right = 2 * i + 1;
while (heap[i] <= heap[left] || heap[i] <= heap[right]) {
if (heap[left] > heap[right]) {
[heap[i], heap[left]] = [heap[left], heap[i]];
i = 2 * i
} else {
[heap[i], heap[right]] = [heap[right], heap[i]];
i = 2 * i + 1;
};
left = 2 * i;
right = 2 * i + 1;
if (heap[left] == undefined || heap[right] == undefined) {
break;
};
};
} else if (heap.length == 2) {
heap.splice(1, 1);
} else {
return null;
};
return smallest;
```

```
};
};
```

stack
sets
queues
binaryST
hash
linkedList
trie
heaps
graphs

Stack*functions: push, pop, peek, length*

Home

```
function bfs(graph, root) {
var nodesLen = {};

for (var i = 0; i < graph.length; i++) {
nodesLen[i] = Infinity;
}
nodesLen[root] = 0;

var queue = [root];
var current;

while (queue.length != 0) {
current = queue.shift();

var curConnected = graph[current];
var neighborIdx = [];
var idx = curConnected.indexOf(1);
while (idx != -1) {
neighborIdx.push(idx);
idx = curConnected.indexOf(1, idx + 1);
}

for (var j = 0; j < neighborIdx.length; j++) {
if (nodesLen[neighborIdx[j]] == Infinity) {
nodesLen[neighborIdx[j]] = nodesLen[current] + 1;
queue.push(neighborIdx[j]);
}
}
}
return nodesLen;
};

var exBFSGraph = [
[0, 1, 1, 1, 0],
[0, 0, 1, 0, 0],
[1, 1, 0, 0, 0],
[0, 0, 0, 1, 0],
[0, 1, 0, 0, 0]
];
console.log(bfs(exBFSGraph, 1));
```