



Open in app

Get started



Published in Better Programming

You have **2** free member-only stories left this month.
[Sign up for Medium and get an extra one](#)



Kingsley Tan

Follow

Oct 8, 2019 · 12 min read · ✨ · 🎧 Listen





Save



8 Common Data Structures in Javascript

Get a better picture of how data structures work

Does this sound familiar: “I started frontend development by completing part-time courses”

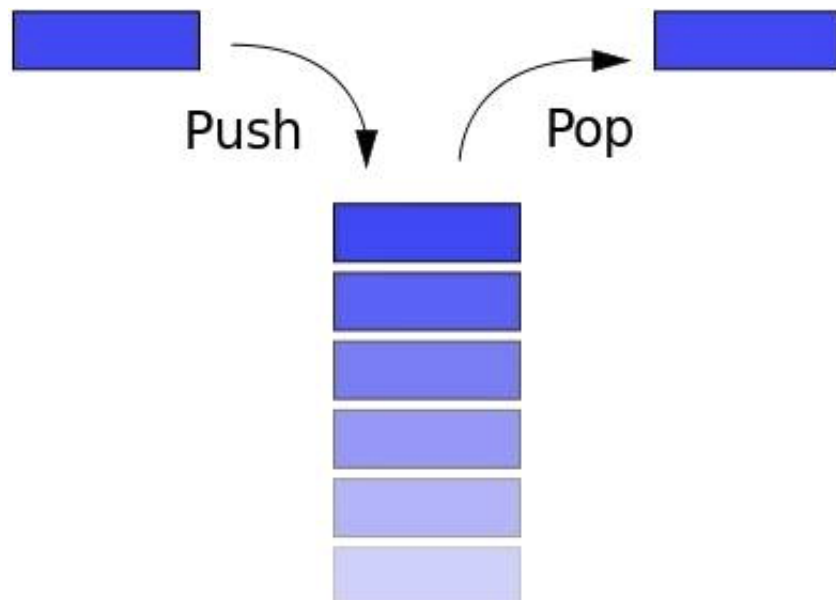
You might be looking to improve your fundamental knowledge of computer science, especially on data st  1.1K |  10 hms. Today we'll go through some common data structures and implement them in JavaScript.



lls!

DATA STRUCTURE

1. Stack



Stack follows the principle of LIFO (Last In First Out). If you stack books, the top book will be taken before the bottom one. Or when you browse on internet, the back button leads you to the most recently browsed page.

Stack has these common methods:

- `push` : input a new element
- `pop` : remove the top element, return the removed element

- `peek` : return the top element
- `length` : return the number of element(s) in Stack

Array in Javascript has the attributes of Stack, but we construct a Stack from scratch by using `function Stack()`

```
function Stack() {
  this.count = 0;
  this.storage = {};

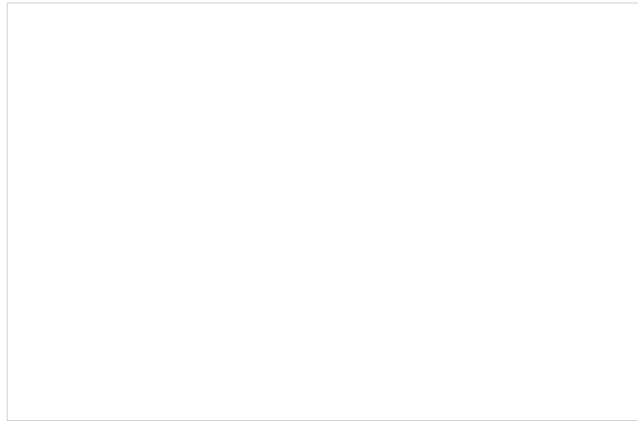
  this.push = function (value) {
    this.storage[this.count] = value;
    this.count++;
  }

  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.peek = function () {
    return this.storage[this.count - 1];
  }

  this.size = function () {
    return this.count;
  }
}
```

2. Queue



Queue is similar to Stack. The only difference is that Queue uses the FIFO principle (First In First Out). In other words, when you queue for bus, the first in the queue will always board first.

Queue has following methods:

- `enqueue` : enter queue, add an element at the end
- `dequeue` : leave queue, remove the front element and return it
- `front` : get the first element
- `isEmpty` : determine whether the queue is empty
- `size` : get the number of element(s) in queue

Array in JavaScript has some attributes of Queue, so we can use array to construct an example for Queue:

```
function Queue() {  
  var 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.isEmpty = function () {
        return collection.length === 0;
    }
    this.size = function () {
        return collection.length;
    }
}

```

Priority Queue

Queue has another advanced version. Allocate each element with priority and it will be sorted according to the priority level:

```

function PriorityQueue() {

    ...

    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]) {
                    collection.splice(i, 0, element);
                    added = true;
                    break;
                }
            }
            if (!added) {
                collection.push(element);
            }
        }
    }
}

```

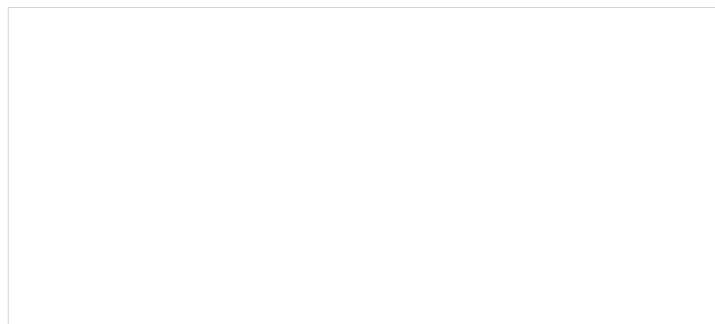
Testing it out:

```
var pQ = new PriorityQueue();
pQ.enqueue([ gannicus , 3]);
pQ.enqueue([ spartacus , 1]);
pQ.enqueue([ crixus , 2]);
pQ.enqueue([ oenomaus , 4]);
pQ.print();
```

Result:

```
[
  [ spartacus , 1 ],
  [ crixus , 2 ],
  [ gannicus , 3 ],
  [ oenomaus , 4 ]
]
```

3. Linked List



Literally, a linked list is a chained data structure, with each node consisting of two pieces of information: the data of the node and the pointer to the next node. Linked list and conventional array are both linear data structures with serialised storage. Of course, they also have differences:

A unilateral linked list normally has following methods:

- `size` : Return the number of node(s)
- `head` : Return the element of the head
- `add` : Add another node in the tail
- `remove` : Remove certain node
- `indexOf` : Return the index of a node
- `elementAt` : Return the node of an index
- `addAt` : Insert a node at a specific index
- `removeAt` : Delete a node at a specific index

```
/** Node in the linked list */  
function Node(element) {  
    // Data in the node  
    this.element = element;  
    // Pointer to the next node  
    this.next = null;  
}  
  
function LinkedList() {  
    var length = 0;  
    var head = 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;
    }
}

```

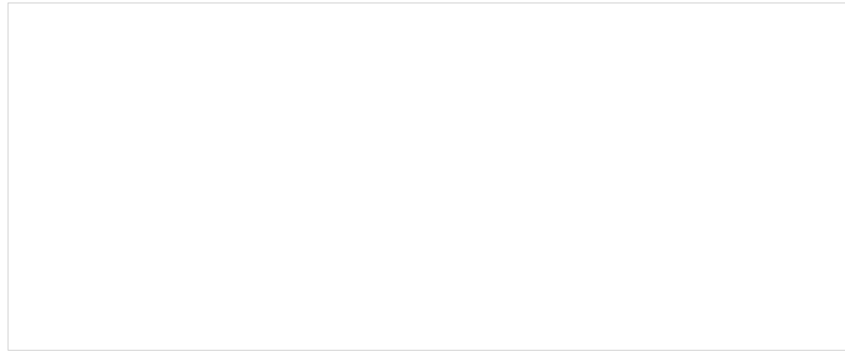


```

        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 = currentIndex.next;
        } else {
            while (currentIndex < index) {
                currentIndex++;
                previousNode = currentNode;
                currentNode = currentNode.next;
            }
            previousNode.next = currentNode.next;
        }
        length--;
        return currentNode.element;
    }
}

```

4. Set



A set is a basic concept in mathematics: a collection of well defined and distinct objects. ES6 introduced the concept of set, which has certain level of similarity with array. However, a set does not allow repeating elements and is not indexed.

A typical set has methods as follows:

- `values` : Return all elements in a set
- `size` : Return the number of elements
- `has` : Determine whether an element exists
- `add` : Insert elements into set
- `remove` : Delete elements from set
- `union` : Return the intersection of two sets
- `difference` : Return the difference of two sets
- `subset` : Determine whether a certain set is a subset of another set

To differentiate `set` in ES6, we declare as `MySet` in following example:

```
function MySet() {  
    var collection = [];  
    this.has = function (element) {  
        return (collection.indexOf(element) !== -1);  
    }  
}
```

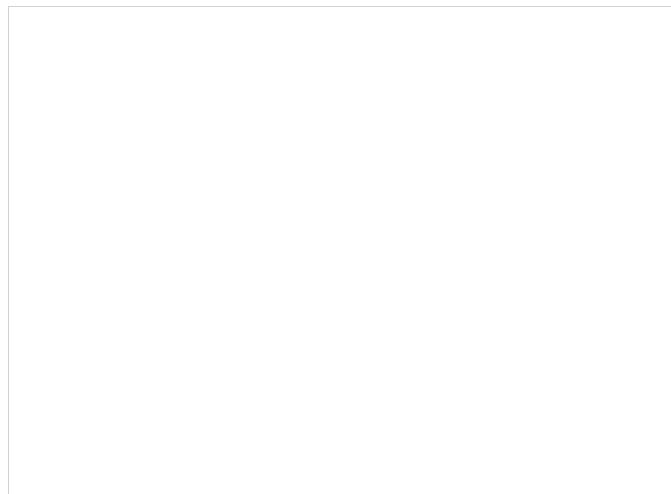
```

this.values = function () {
    return collection;
}
this.size = function () {
    return collection.length;
}
this.add = function (element) {
    if (!this.has(element)) {
        collection.push(element);
        return true;
    }
    return false;
}
this.remove = function (element) {
    if (this.has(element)) {
        index = collection.indexOf(element);
        collection.splice(index, 1);
        return true;
    }
    return false;
}
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.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.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.subset = function (otherSet) {
        var firstSet = this.values();
        return firstSet.every(function (value) {
            return otherSet.has(value);
        });
    }
}
```

5. Hast table



A hash table is a key-value data structure. Due to the lightning speed of querying a value through key, it is commonly used in Map, Dictionary or Object data structures. As shown in the graph above, the hash table uses a **hash function** to convert keys into a list of numbers, and these numbers serve as the values of corresponding keys. To get value using key is dashingly fast, time complexity can achieve $O(1)$. The same keys must return the same values — this is the basis of the hash function.

The hash table has the following methods:

- **add** : Add a key-value pair
- **remove** : Delete a key-value pair

- **lookup** : Find a corresponding value using a key

An example of a simplified Hash Table in Javascript:

```
function hash(string, max) {
  var hash = 0;
  for (var i = 0; i < string.length; i++) {
    hash += string.charCodeAt(i);
  }
  return hash % max;
}

function HashTable() {
  let storage = [];
  const storageLimit = 4;

  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]; 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];
        }
      }
    }
  }
}

```

6. Tree

Tree data structure is a multi-layer structure. It is also a non-linear data structure, compared to Array, Stack, and Queue. This structure is highly efficient during insert and search operations. Let's take a look at some concepts of tree data structure:

- **root** : Root node of a tree, no parent node for root
- **parent node** : Direct node of the upper layer, only has one
- **child node** : Direct node(s) of the lower layer, can have multiple
- **siblings** : Share the same parent node
- **leaf** : Node with no child
- **Edge** : Branch or link between nodes
- **Path** : The edges from a starting node to the target node
- **Height of Node** : Number of edges of the longest path of a specific node to leaf node
- **Height of Tree** : Number of edges of the longest path of the root node to the leaf node
- **Depth of Node** : Number of edges from root node to specific node
- **Degree of Node** : Number of child nodes

Here's an example of a binary search tree here. Each node has a maximum of two nodes with the left node being smaller than the current node and the right node being bigger than the current node:



Common methods in Binary Search Tree:

- `add` : Insert a node into the tree
- `findMin` : Get the minimum node
- `findMax` : Get the maximum node
- `find` : Search a specific node
- `isPresent` : Determine the existence of a certain node
- `remove` : Delete a node from the tree

Example in JavaScript:

```
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) {
            // no child node
            if (node.left == null && node.right == null) {
                return null;
            }
            // no left node
            if (node.left == null) {
                return node.right;
            }
            // no right node
            if (node.right == null) {
                return node.left;
            }
            // has 2 child nodes
            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);
}
}

```

Testing it out:

```

const bst = new BST();
bst.add(4);
bst.add(2);
bst.add(6);
bst.add(1);
bst.add(3);
bst.add(5);
bst.add(7);
bst.remove(4);
console.log(bst.findMin());
console.log(bst.findMax());
bst.remove(7);
console.log(bst.findMax());
console.log(bst.isPresent(4));

```

Result:

```

1
7
6
false

```

7. Trie (pronounced “try”)

Trie, or “Prefix Tree”, is also a type of search tree. Trie stores the data step-by-step — each node in the tree represents a step. Trie is used in storing vocabulary so it can be quickly searched, especially for an auto-complete function.

Each node in Trie has an alphabet — following the branch can form a complete word. It also comprises a boolean indicator to show whether is this the last alphabet.

Trie has following methods:

- **add** : Insert a word into the dictionary tree

- `isWord`: Determine whether the tree consists of certain word
- `print`: Return all words in the tree

```

/** Node in Trie */
function Node() {
  this.keys = new Map();
  this.end = false;
  this.setEnd = function () {
    this.end = true;
  };
  this.isEnd = function () {
    return this.end;
  }
}

function Trie() {
  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 = this.root,

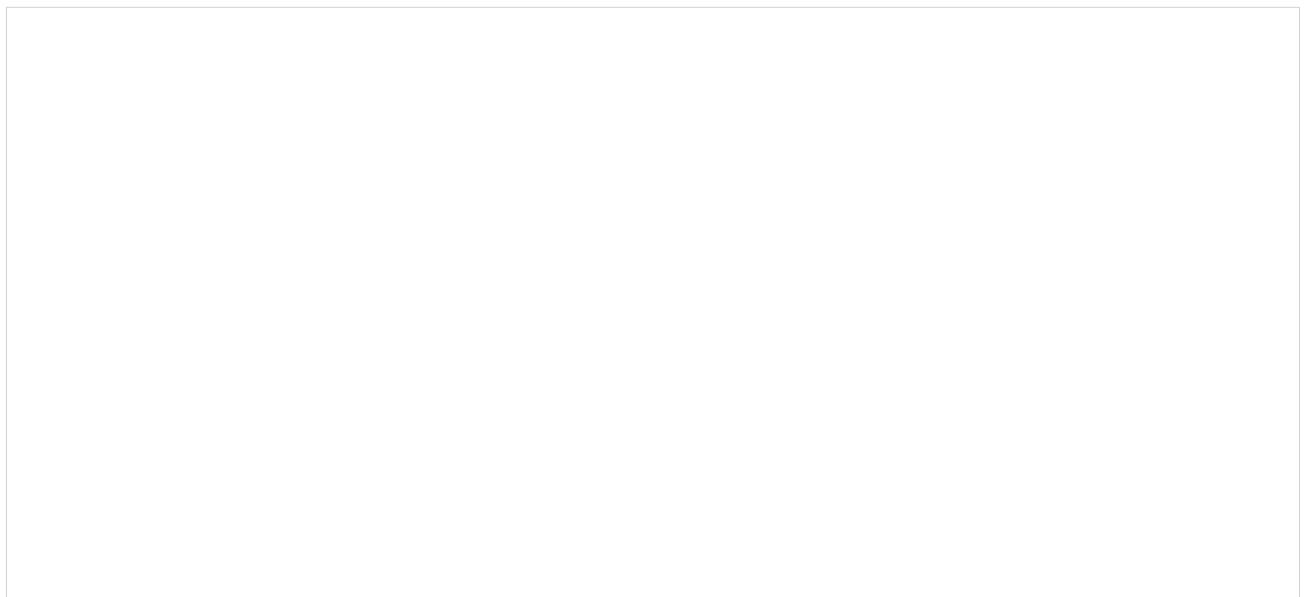
```

```

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 : null;
}
}

```

8. Graph



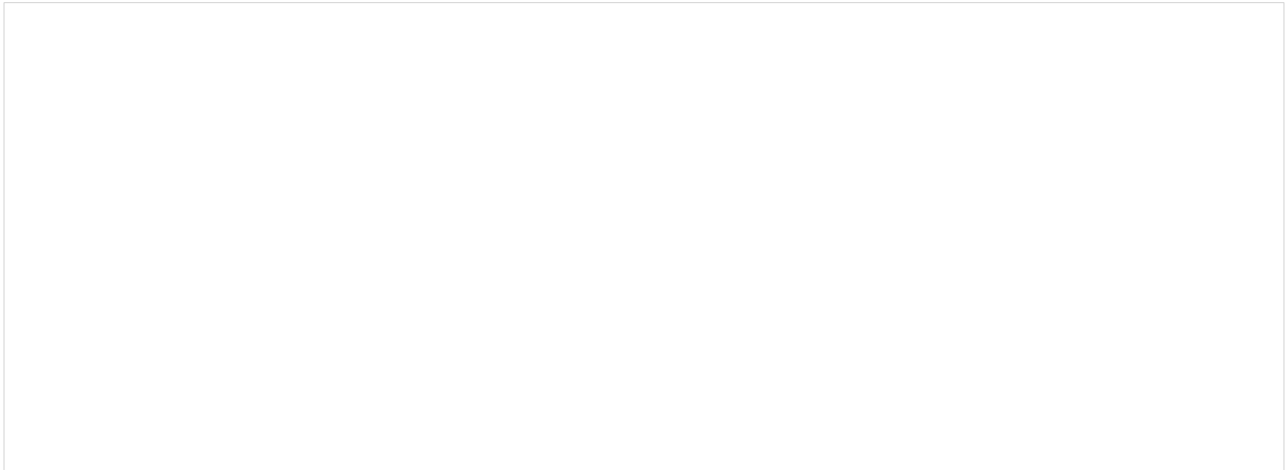
Graph, sometimes known as network, refers to sets of nodes with linkages (or edges). It could be further divided into two groups (ie. directed graphs and undirected graphs), according to whether the linkages have direction. Graph is widely used in our lives — to calculate the best route in navigation

apps, or to recommended friends in social media, to take two examples.

Graph has two types of presentation:

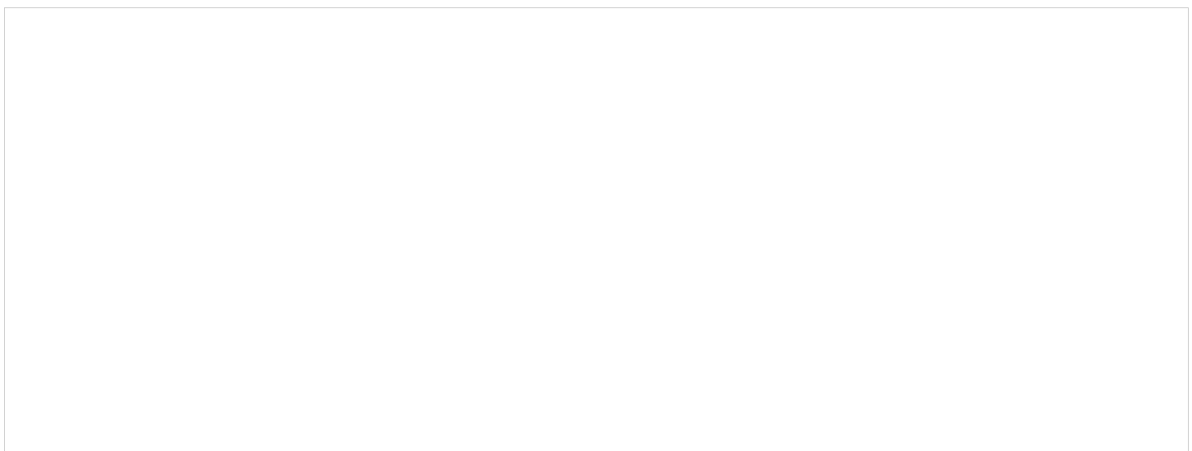
Adjacency List

In this method, we list all the possible nodes on the left and show the connected nodes on the right.



Adjacency Matrix

Adjacency matrix shows nodes in row and column, intersections of the row and column interpret the relationship between nodes, 0 means not linked, 1 means linked, >1 means different weightage.



To query for nodes in graph, one must search through the entire tree network with either the Breath-First-Search (BFS) method or the Depth-First-Search (DFS) method.

Let's see an example of BFS in Javascript:

```
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;
}
```

Testing it out:

```
var graph = [
  [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(graph, 1));
```

Result:

```
{  
  0: 2,  
  1: 0,  
  2: 1,  
  3: 3,  
  4: Infinity  
}
```

That's it — we have covered all the common data structures and given examples in JavaScript. This should give you a better picture of how data structures work in computers. Happy coding!

Sign up for Coffee Bytes

By Better Programming

A newsletter covering the best programming articles published across Medium [Take a look.](#)

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



Get this newsletter



[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

