

```
1 class MyArray {
2   constructor() {
3     this.array = [];
4   }
5
6   add(data) {
7     this.array.push(data);
8   }
9
10  remove(data) {
11    this.array = this.array.filter(current => current !== data);
12  }
13
14  search(data) {
15    const foundIndex = this.array.indexOf(data);
16    if(~foundIndex) {
17      return foundIndex;
18    }
19
20    return null;
21  }
22
23  getAtIndex(index) {
24    return this.array[index];
25  }
26
27  length() {
28    return this.array.length;
29  }
30
31  print() {
32    console.log(this.array.join(' '));
33  }
34 }
35
36 const array = new MyArray();
37 array.add(1);
38 array.add(2);
39 array.add(3);
40 array.add(4);
41 array.print(); // => 1 2 3 4
42 console.log('search 3 gives index 2:', array.search(3)); // => 2
43 console.log('getAtIndex 2 gives 3:', array.getAtIndex(2)); // => 3
44 console.log('length is 4:', array.length()); // => 4
45 array.remove(3);
46 array.print(); // => 1 2 4
47 array.add(5);
48 array.add(5);
49 array.print(); // => 1 2 4 5 5
50 array.remove(5);
51 array.print(); // => 1 2 4
52
```

```
1 function Node(data) {
2   this.data = data;
3   this.left = null;
4   this.right = null;
5 }
6
7 class BinarySearchTree {
8   constructor() {
9     this.root = null;
10  }
11
12  add(data) {
13    const node = new Node(data);
14    if(!this.root) {
15      this.root = node;
16    } else {
17      let current = this.root;
18      while(current) {
19        if(node.data < current.data) {
20          if(!current.left) {
21            current.left = node;
22            break;
23          }
24          current = current.left;
25        } else if (node.data > current.data) {
26          if(!current.right) {
27            current.right = node;
28            break;
29          }
30          current = current.right;
31        } else {
32          break;
33        }
34      }
35    }
36  }
37
38  remove(data) {
39    const that = this;
40    const removeNode = (node, data) => {
41      if(!node) {
42        return null;
43      }
44      if(data === node.data) {
45        if(!node.left && !node.right) {
46          return null;
47        }
48        if(!node.left) {
49          return node.right;
50        }
51        if(!node.right) {
52          return node.left;
53        }
54        // 2 children
55        const temp = that.getMin(node.right);
56        node.data = temp;
57        node.right = removeNode(node.right, temp);
58        return node;
59      } else if(data < node.data) {
60        node.left = removeNode(node.left, data);
```

```
61     return node;
62   } else {
63     node.right = removeNode(node.right, data);
64     return node;
65   }
66 };
67 this.root = removeNode(this.root, data);
68 }
69
70 contains(data) {
71   let current = this.root;
72   while(current) {
73     if(data === current.data) {
74       return true;
75     }
76     if(data < current.data) {
77       current = current.left;
78     } else {
79       current = current.right;
80     }
81   }
82   return false;
83 }
84
85 _preOrder(node, fn) {
86   if(node) {
87     if(fn) {
88       fn(node);
89     }
90     this._preOrder(node.left, fn);
91     this._preOrder(node.right, fn);
92   }
93 }
94
95 _inOrder(node, fn) {
96   if(node) {
97     this._inOrder(node.left, fn);
98     if(fn) {
99       fn(node);
100   }
101   this._inOrder(node.right, fn);
102 }
103 }
104
105 _postOrder(node, fn) {
106   if(node) {
107     this._postOrder(node.left, fn);
108     this._postOrder(node.right, fn);
109     if(fn) {
110       fn(node);
111     }
112   }
113 }
114
115 traverseDFS(fn, method) {
116   const current = this.root;
117   if(method) {
118     this[`_${method}`](current, fn);
119   } else {
120     this._preOrder(current, fn);
```

```
121     }
122   }
123
124   traverseBFS(fn) {
125     this.queue = [];
126     this.queue.push(this.root);
127     while(this.queue.length) {
128       const node = this.queue.shift();
129       if(fn) {
130         fn(node);
131       }
132       if(node.left) {
133         this.queue.push(node.left);
134       }
135       if(node.right) {
136         this.queue.push(node.right);
137       }
138     }
139   }
140
141   print() {
142     if(!this.root) {
143       return console.log('No root node found');
144     }
145     const newline = new Node('|');
146     const queue = [this.root, newline];
147     let string = '';
148     while(queue.length) {
149       const node = queue.shift();
150       string += `${node.data.toString()} `;
151       if(node === newline && queue.length) {
152         queue.push(newline);
153       }
154       if(node.left) {
155         queue.push(node.left);
156       }
157       if(node.right) {
158         queue.push(node.right);
159       }
160     }
161     console.log(string.slice(0, -2).trim());
162   }
163
164   printByLevel() {
165     if(!this.root) {
166       return console.log('No root node found');
167     }
168     const newline = new Node('\n');
169     const queue = [this.root, newline];
170     let string = '';
171     while(queue.length) {
172       const node = queue.shift();
173       string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
174       if(node === newline && queue.length) {
175         queue.push(newline);
176       }
177       if(node.left) {
178         queue.push(node.left);
179       }
180       if(node.right) {
```

```
181     queue.push(node.right);
182   }
183 }
184 console.log(string.trim());
185 }
186
187 getMin(node) {
188   if(!node) {
189     node = this.root;
190   }
191   while(node.left) {
192     node = node.left;
193   }
194   return node.data;
195 }
196
197 getMax(node) {
198   if(!node) {
199     node = this.root;
200   }
201   while(node.right) {
202     node = node.right;
203   }
204   return node.data;
205 }
206
207 _getHeight(node) {
208   if(!node) {
209     return -1;
210   }
211   const left = this._getHeight(node.left);
212   const right = this._getHeight(node.right);
213   return Math.max(left, right) + 1;
214 }
215
216 getHeight(node) {
217   if(!node) {
218     node = this.root;
219   }
220   return this._getHeight(node);
221 }
222
223 _isBalanced(node) {
224   if(!node) {
225     return true;
226   }
227   const heightLeft = this._getHeight(node.left);
228   const heightRight = this._getHeight(node.right);
229   const diff = Math.abs(heightLeft - heightRight);
230   if(diff > 1) {
231     return false;
232   } else {
233     return this._isBalanced(node.left) && this._isBalanced(node.right);
234   }
235 }
236
237 isBalanced(node) {
238   if(!node) {
239     node = this.root;
240   }
```

```
241     return this._isBalanced(node);
242 }
243
244 _checkHeight(node) {
245     if(!node) {
246         return 0;
247     }
248     const left = this._checkHeight(node.left);
249     if(left === -1) {
250         return -1;
251     }
252     const right = this._checkHeight(node.right);
253     if(right === -1) {
254         return -1;
255     }
256     const diff = Math.abs(left - right);
257     if(diff > 1) {
258         return -1;
259     } else {
260         return Math.max(left, right) + 1;
261     }
262 }
263
264 isBalancedOptimized(node) {
265     if(!node) {
266         node = this.root;
267     }
268     if(!node) {
269         return true;
270     }
271     if(this._checkHeight(node) === -1) {
272         return false;
273     } else {
274         return true;
275     }
276 }
277 }
278
279 const binarySearchTree = new BinarySearchTree();
280 binarySearchTree.add(5);
281 binarySearchTree.add(3);
282 binarySearchTree.add(7);
283 binarySearchTree.add(2);
284 binarySearchTree.add(4);
285 binarySearchTree.add(4);
286 binarySearchTree.add(6);
287 binarySearchTree.add(8);
288 binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8
289 binarySearchTree.printByLevel(); // => 5 \n 3 7 \n 2 4 6 8
290 console.log('--- DFS inOrder');
291 binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'inOrder'); // => 2
3 4 5 6 7 8
292 console.log('--- DFS preOrder');
293 binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'preOrder'); // =>
5 3 2 4 7 6 8
294 console.log('--- DFS postOrder');
295 binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'postOrder'); // =>
2 4 3 6 8 7 5
296 console.log('--- BFS');
```

```
297 binarySearchTree.traverseBFS(node => { console.log(node.data); }); // => 5 3 7 2 4 6
8
298 console.log('min is 2:', binarySearchTree.getMin()); // => 2
299 console.log('max is 8:', binarySearchTree.getMax()); // => 8
300 console.log('tree contains 3 is true:', binarySearchTree.contains(3)); // => true
301 console.log('tree contains 9 is false:', binarySearchTree.contains(9)); // => false
302 console.log('tree height is 2:', binarySearchTree.getHeight()); // => 2
303 console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
304 binarySearchTree.remove(11); // remove non existing node
305 binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8
306 binarySearchTree.remove(5); // remove 5, 6 goes up
307 binarySearchTree.print(); // => 6 | 3 7 | 2 4 8
308 binarySearchTree.remove(7); // remove 7, 8 goes up
309 binarySearchTree.print(); // => 6 | 3 8 | 2 4
310 binarySearchTree.remove(8); // remove 8, the tree becomes unbalanced
311 binarySearchTree.print(); // => 6 | 3 | 2 4
312 console.log('tree is balanced is false:', binarySearchTree.isBalanced()); // => true
313 binarySearchTree.remove(4);
314 binarySearchTree.remove(2);
315 binarySearchTree.remove(3);
316 binarySearchTree.remove(6);
317 binarySearchTree.print(); // => 'No root node found'
318 binarySearchTree.printByLevel(); // => 'No root node found'
319 console.log('tree height is -1:', binarySearchTree.getHeight()); // => -1
320 console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
321 console.log('---');
322 binarySearchTree.add(10);
323 console.log('tree height is 0:', binarySearchTree.getHeight()); // => 0
324 console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
325 binarySearchTree.add(6);
326 binarySearchTree.add(14);
327 binarySearchTree.add(4);
328 binarySearchTree.add(8);
329 binarySearchTree.add(12);
330 binarySearchTree.add(16);
331 binarySearchTree.add(3);
332 binarySearchTree.add(5);
333 binarySearchTree.add(7);
334 binarySearchTree.add(9);
335 binarySearchTree.add(11);
336 binarySearchTree.add(13);
337 binarySearchTree.add(15);
338 binarySearchTree.add(17);
339 binarySearchTree.print(); // => 10 | 6 14 | 4 8 12 16 | 3 5 7 9 11 13 15 17
340 binarySearchTree.remove(10); // remove 10, 11 goes up
341 binarySearchTree.print(); // => 11 | 6 14 | 4 8 12 16 | 3 5 7 9 x 13 15 17
342 binarySearchTree.remove(12); // remove 12; 13 goes up
343 binarySearchTree.print(); // => 11 | 6 14 | 4 8 13 16 | 3 5 7 9 x x 15 17
344 console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
345 console.log('tree is balanced optimized is true:',
  binarySearchTree.isBalancedOptimized()); // => true
346 binarySearchTree.remove(13); // remove 13, 13 has no children so nothing changes
347 binarySearchTree.print(); // => 11 | 6 14 | 4 8 x 16 | 3 5 7 9 x x 15 17
348 console.log('tree is balanced is false:', binarySearchTree.isBalanced()); // => false
349 console.log('tree is balanced optimized is false:',
  binarySearchTree.isBalancedOptimized()); // => false
350
```

```
1 function Node(data) {
2   this.data = data;
3   this.previous = null;
4   this.next = null;
5 }
6
7 class DoublyLinkedList {
8   constructor() {
9     this.head = null;
10    this.tail = null;
11    this.numberOfValues = 0;
12  }
13
14  add(data) {
15    const node = new Node(data);
16    if(!this.head) {
17      this.head = node;
18      this.tail = node;
19    } else {
20      node.previous = this.tail;
21      this.tail.next = node;
22      this.tail = node;
23    }
24    this.numberOfValues++;
25  }
26
27  remove(data) {
28    let current = this.head;
29    while(current) {
30      if(current.data === data) {
31        if(current === this.head && current === this.tail) {
32          this.head = null;
33          this.tail = null;
34        } else if(current === this.head) {
35          this.head = this.head.next;
36          this.head.previous = null;
37        } else if(current === this.tail) {
38          this.tail = this.tail.previous;
39          this.tail.next = null;
40        } else {
41          current.previous.next = current.next;
42          current.next.previous = current.previous;
43        }
44        this.numberOfValues--;
45      }
46      current = current.next;
47    }
48  }
49
50  insertAfter(data, toNodeData) {
51    let current = this.head;
52    while(current) {
53      if(current.data === toNodeData) {
54        const node = new Node(data);
55        if(current === this.tail) {
56          this.add(data);
57        } else {
58          current.next.previous = node;
59          node.previous = current;
60          node.next = current.next;
```



```
61         current.next = node;
62         this.numberOfValues++;
63     }
64 }
65     current = current.next;
66 }
67 }
68
69 traverse(fn) {
70     let current = this.head;
71     while(current) {
72         if(fn) {
73             fn(current);
74         }
75         current = current.next;
76     }
77 }
78
79 traverseReverse(fn) {
80     let current = this.tail;
81     while(current) {
82         if(fn) {
83             fn(current);
84         }
85         current = current.previous;
86     }
87 }
88
89 length() {
90     return this.numberOfValues;
91 }
92
93 print() {
94     let string = '';
95     let current = this.head;
96     while(current) {
97         string += `${current.data} `;
98         current = current.next;
99     }
100     console.log(string.trim());
101 }
102 }
103
104 const doublyLinkedList = new DoublyLinkedList();
105 doublyLinkedList.print(); // => ''
106 doublyLinkedList.add(1);
107 doublyLinkedList.add(2);
108 doublyLinkedList.add(3);
109 doublyLinkedList.add(4);
110 doublyLinkedList.print(); // => 1 2 3 4
111 console.log('length is 4:', doublyLinkedList.length()); // => 4
112 doublyLinkedList.remove(3); // remove value
113 doublyLinkedList.print(); // => 1 2 4
114 doublyLinkedList.remove(9); // remove non existing value
115 doublyLinkedList.print(); // => 1 2 4
116 doublyLinkedList.remove(1); // remove head
117 doublyLinkedList.print(); // => 2 4
118 doublyLinkedList.remove(4); // remove tail
119 doublyLinkedList.print(); // => 2
120 console.log('length is 1:', doublyLinkedList.length()); // => 1
```

```
121 doublyLinkedList.remove(2); // remove tail, the list should be empty
122 doublyLinkedList.print(); // => ''
123 console.log('length is 0:', doublyLinkedList.length()); // => 0
124 doublyLinkedList.add(2);
125 doublyLinkedList.add(6);
126 doublyLinkedList.print(); // => 2 6
127 doublyLinkedList.insertAfter(3, 2);
128 doublyLinkedList.print(); // => 2 3 6
129 doublyLinkedList.traverseReverse(node => { console.log(node.data); });
130 doublyLinkedList.insertAfter(4, 3);
131 doublyLinkedList.print(); // => 2 3 4 6
132 doublyLinkedList.insertAfter(5, 9); // insertAfter a non existing node
133 doublyLinkedList.print(); // => 2 3 4 6
134 doublyLinkedList.insertAfter(5, 4);
135 doublyLinkedList.insertAfter(7, 6); // insertAfter the tail
136 doublyLinkedList.print(); // => 2 3 4 5 6 7
137 doublyLinkedList.add(8); // add node with normal method
138 doublyLinkedList.print(); // => 2 3 4 5 6 7 8
139 console.log('length is 7:', doublyLinkedList.length()); // => 7
140 doublyLinkedList.traverse(node => { node.data = node.data + 10; });
141 doublyLinkedList.print(); // => 12 13 14 15 16 17 18
142 doublyLinkedList.traverse(node => { console.log(node.data); }); // => 12 13 14 15 16
143 console.log('length is 7:', doublyLinkedList.length()); // => 7
144 doublyLinkedList.traverseReverse(node => { console.log(node.data); }); // => 18 17 16
145 doublyLinkedList.print(); // => 12 13 14 15 16 17 18
146 console.log('length is 7:', doublyLinkedList.length()); // => 7
147
```

```
1 class Graph {
2   constructor() {
3     this.vertices = [];
4     this.edges = [];
5     this.numberOfEdges = 0;
6   }
7
8   addVertex(vertex) {
9     this.vertices.push(vertex);
10    this.edges[vertex] = [];
11  }
12
13  removeVertex(vertex) {
14    const index = this.vertices.indexOf(vertex);
15    if(~index) {
16      this.vertices.splice(index, 1);
17    }
18    while(this.edges[vertex].length) {
19      const adjacentVertex = this.edges[vertex].pop();
20      this.removeEdge(adjacentVertex, vertex);
21    }
22  }
23
24  addEdge(vertex1, vertex2) {
25    this.edges[vertex1].push(vertex2);
26    this.edges[vertex2].push(vertex1);
27    this.numberOfEdges++;
28  }
29
30  removeEdge(vertex1, vertex2) {
31    const index1 = this.edges[vertex1] ? this.edges[vertex1].indexOf(vertex2) : -1;
32    const index2 = this.edges[vertex2] ? this.edges[vertex2].indexOf(vertex1) : -1;
33    if(~index1) {
34      this.edges[vertex1].splice(index1, 1);
35      this.numberOfEdges--;
36    }
37    if(~index2) {
38      this.edges[vertex2].splice(index2, 1);
39    }
40  }
41
42  size() {
43    return this.vertices.length;
44  }
45
46  relations() {
47    return this.numberOfEdges;
48  }
49
50  traverseDFS(vertex, fn) {
51    if(!~this.vertices.indexOf(vertex)) {
52      return console.log('Vertex not found');
53    }
54    const visited = [];
55    this._traverseDFS(vertex, visited, fn);
56  }
57
58  _traverseDFS(vertex, visited, fn) {
59    visited[vertex] = true;
60    if(this.edges[vertex] !== undefined) {
```

```
61     fn(vertex);
62   }
63   for(let i = 0; i < this.edges[vertex].length; i++) {
64     if(!visited[this.edges[vertex][i]]) {
65       this._traverseDFS(this.edges[vertex][i], visited, fn);
66     }
67   }
68 }
69
70 traverseBFS(vertex, fn) {
71   if(!~this.vertices.indexOf(vertex)) {
72     return console.log('Vertex not found');
73   }
74   const queue = [];
75   queue.push(vertex);
76   const visited = [];
77   visited[vertex] = true;
78
79   while(queue.length) {
80     vertex = queue.shift();
81     fn(vertex);
82     for(let i = 0; i < this.edges[vertex].length; i++) {
83       if(!visited[this.edges[vertex][i]]) {
84         visited[this.edges[vertex][i]] = true;
85         queue.push(this.edges[vertex][i]);
86       }
87     }
88   }
89 }
90
91 pathFromTo(vertexSource, vertexDestination) {
92   if(!~this.vertices.indexOf(vertexSource)) {
93     return console.log('Vertex not found');
94   }
95   const queue = [];
96   queue.push(vertexSource);
97   const visited = [];
98   visited[vertexSource] = true;
99   const paths = [];
100
101   while(queue.length) {
102     const vertex = queue.shift();
103     for(let i = 0; i < this.edges[vertex].length; i++) {
104       if(!visited[this.edges[vertex][i]]) {
105         visited[this.edges[vertex][i]] = true;
106         queue.push(this.edges[vertex][i]);
107         // save paths between vertices
108         paths[this.edges[vertex][i]] = vertex;
109       }
110     }
111   }
112   if(!visited[vertexDestination]) {
113     return undefined;
114   }
115
116   const path = [];
117   for(var j = vertexDestination; j !== vertexSource; j = paths[j]) {
118     path.push(j);
119   }
120   path.push(j);
```

```

121     return path.reverse().join('-');
122 }
123
124 print() {
125     console.log(this.vertices.map(function(vertex) {
126         return (`${vertex} -> ${this.edges[vertex].join(', ')}').trim();
127     }, this).join(' | '));
128 }
129 }
130
131 const graph = new Graph();
132 graph.addVertex(1);
133 graph.addVertex(2);
134 graph.addVertex(3);
135 graph.addVertex(4);
136 graph.addVertex(5);
137 graph.addVertex(6);
138 graph.print(); // 1 -> | 2 -> | 3 -> | 4 -> | 5 -> | 6 ->
139 graph.addEdge(1, 2);
140 graph.addEdge(1, 5);
141 graph.addEdge(2, 3);
142 graph.addEdge(2, 5);
143 graph.addEdge(3, 4);
144 graph.addEdge(4, 5);
145 graph.addEdge(4, 6);
146 graph.print(); // 1 -> 2, 5 | 2 -> 1, 3, 5 | 3 -> 2, 4 | 4 -> 3, 5, 6 | 5 -> 1, 2, 4
    | 6 -> 4
147 console.log('graph size (number of vertices):', graph.size()); // => 6
148 console.log('graph relations (number of edges):', graph.relations()); // => 7
149 graph.traverseDFS(1, vertex => { console.log(vertex); }); // => 1 2 3 4 5 6
150 console.log('---');
151 graph.traverseBFS(1, vertex => { console.log(vertex); }); // => 1 2 5 3 4 6
152 graph.traverseDFS(0, vertex => { console.log(vertex); }); // => 'Vertex not found'
153 graph.traverseBFS(0, vertex => { console.log(vertex); }); // => 'Vertex not found'
154 console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-5-1
155 console.log('path from 3 to 5:', graph.pathFromTo(3, 5)); // => 3-2-5
156 graph.removeEdge(1, 2);
157 graph.removeEdge(4, 5);
158 graph.removeEdge(10, 11);
159 console.log('graph relations (number of edges):', graph.relations()); // => 5
160 console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-3-2-5-1
161 graph.addEdge(1, 2);
162 graph.addEdge(4, 5);
163 console.log('graph relations (number of edges):', graph.relations()); // => 7
164 console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-5-1
165 graph.removeVertex(5);
166 console.log('graph size (number of vertices):', graph.size()); // => 5
167 console.log('graph relations (number of edges):', graph.relations()); // => 4
168 console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-3-2-1
169

```

```
1 class HashTable {
2   constructor(size) {
3     this.values = {};
4     this.numberOfValues = 0;
5     this.size = size;
6   }
7
8   add(key, value) {
9     const hash = this.calculateHash(key);
10    if(!this.values.hasOwnProperty(hash)) {
11      this.values[hash] = {};
12    }
13    if(!this.values[hash].hasOwnProperty(key)) {
14      this.numberOfValues++;
15    }
16    this.values[hash][key] = value;
17  }
18
19  remove(key) {
20    const hash = this.calculateHash(key);
21    if(this.values.hasOwnProperty(hash) && this.values[hash].hasOwnProperty(key)) {
22      delete this.values[hash][key];
23      this.numberOfValues--;
24    }
25  }
26
27  calculateHash(key) {
28    return key.toString().length % this.size;
29  }
30
31  search(key) {
32    const hash = this.calculateHash(key);
33    if(this.values.hasOwnProperty(hash) && this.values[hash].hasOwnProperty(key)) {
34      return this.values[hash][key];
35    } else {
36      return null;
37    }
38  }
39
40  length() {
41    return this.numberOfValues;
42  }
43
44  print() {
45    let string = '';
46    for(const value in this.values) {
47      for(const key in this.values[value]) {
48        string += `${this.values[value][key]} `;
49      }
50    }
51    console.log(string.trim());
52  }
53 }
54
55 const hashTable = new HashTable(3);
56 hashTable.add('first', 1);
57 hashTable.add('second', 2);
58 hashTable.add('third', 3);
59 hashTable.add('fourth', 4);
60 hashTable.add('fifth', 5);
```

```
61 hashTable.print(); // => 2 4 1 3 5
62 console.log('length gives 5:', hashTable.length()); // => 5
63 console.log('search second gives 2:', hashTable.search('second')); // => 2
64 hashTable.remove('fourth');
65 hashTable.remove('first');
66 hashTable.print(); // => 2 3 5
67 console.log('length gives 3:', hashTable.length()); // => 3
68
```

```
1 class Queue {
2   constructor() {
3     this.queue = [];
4   }
5
6   enqueue(value) {
7     this.queue.push(value);
8   }
9
10  dequeue() {
11    return this.queue.shift();
12  }
13
14  peek() {
15    return this.queue[0];
16  }
17
18  length() {
19    return this.queue.length;
20  }
21
22  print() {
23    console.log(this.queue.join(' '));
24  }
25 }
26
27 const queue = new Queue();
28 queue.enqueue(1);
29 queue.enqueue(2);
30 queue.enqueue(3);
31 queue.print(); // => 1 2 3
32 console.log('length is 3:', queue.length()); // => 3
33 console.log('peek is 1:', queue.peek()); // => 3
34 console.log('dequeue is 1:', queue.dequeue()); // => 1
35 queue.print(); // => 2 3
36 console.log('dequeue is 2:', queue.dequeue()); // => 2
37 console.log('length is 1:', queue.length()); // => 1
38 console.log('dequeue is 3:', queue.dequeue()); // => 3
39 queue.print(); // => ''
40 console.log('peek is undefined:', queue.peek()); // => undefined
41 console.log('dequeue is undefined:', queue.dequeue()); // => undefined
42
```



```
1 class Set {
2   constructor() {
3     this.values = [];
4     this.numberOfValues = 0;
5   }
6
7   add(value) {
8     if(!~this.values.indexOf(value)) {
9       this.values.push(value);
10      this.numberOfValues++;
11    }
12  }
13
14  remove(value) {
15    const index = this.values.indexOf(value);
16    if(~index) {
17      this.values.splice(index, 1);
18      this.numberOfValues--;
19    }
20  }
21
22  contains(value) {
23    return this.values.indexOf(value) !== -1;
24  }
25
26  union(set) {
27    const newSet = new Set();
28    set.values.forEach(value => {
29      newSet.add(value);
30    });
31    this.values.forEach(value => {
32      newSet.add(value);
33    });
34    return newSet;
35  }
36
37  intersect(set) {
38    const newSet = new Set();
39    this.values.forEach(value => {
40      if(set.contains(value)) {
41        newSet.add(value);
42      }
43    });
44    return newSet;
45  }
46
47  difference(set) {
48    const newSet = new Set();
49    this.values.forEach(value => {
50      if(!set.contains(value)) {
51        newSet.add(value);
52      }
53    });
54    return newSet;
55  }
56
57  isSubset(set) {
58    return set.values.every(function(value) {
59      return this.contains(value);
60    }, this);
61  }
62}
```

```
61 }
62
63 length() {
64     return this.numberOfValues;
65 }
66
67 print() {
68     console.log(this.values.join(' '));
69 }
70 }
71
72 const set = new Set();
73 set.add(1);
74 set.add(2);
75 set.add(3);
76 set.add(4);
77 set.print(); // => 1 2 3 4
78 set.remove(3);
79 set.print(); // => 1 2 4
80 console.log('contains 4 is true:', set.contains(4)); // => true
81 console.log('contains 3 is false:', set.contains(3)); // => false
82 console.log('---');
83 const set1 = new Set();
84 set1.add(1);
85 set1.add(2);
86 const set2 = new Set();
87 set2.add(2);
88 set2.add(3);
89 const set3 = set2.union(set1);
90 set3.print(); // => 1 2 3
91 const set4 = set2.intersect(set1);
92 set4.print(); // => 2
93 const set5 = set.difference(set3); // 1 2 4 diff 1 2 3
94 set5.print(); // => 4
95 const set6 = set3.difference(set); // 1 2 3 diff 1 2 4
96 set6.print(); // => 3
97 console.log('set1 subset of set is true:', set.isSubset(set1)); // => true
98 console.log('set2 subset of set is false:', set.isSubset(set2)); // => false
99 console.log('set1 length gives 2:', set1.length()); // => 2
100 console.log('set3 length gives 3:', set3.length()); // => 3
101
```

```
1 function Node(data) {
2   this.data = data;
3   this.next = null;
4 }
5
6 class SinglyLinkedList {
7   constructor() {
8     this.head = null;
9     this.tail = null;
10    this.numberOfValues = 0;
11  }
12
13  add(data) {
14    const node = new Node(data);
15    if(!this.head) {
16      this.head = node;
17      this.tail = node;
18    } else {
19      this.tail.next = node;
20      this.tail = node;
21    }
22    this.numberOfValues++;
23  }
24
25  remove(data) {
26    let previous = this.head;
27    let current = this.head;
28    while(current) {
29      if(current.data === data) {
30        if(current === this.head) {
31          this.head = this.head.next;
32        }
33        if(current === this.tail) {
34          this.tail = previous;
35        }
36        previous.next = current.next;
37        this.numberOfValues--;
38      } else {
39        previous = current;
40      }
41      current = current.next;
42    }
43  }
44
45  insertAfter(data, toNodeData) {
46    let current = this.head;
47    while(current) {
48      if(current.data === toNodeData) {
49        const node = new Node(data);
50        if(current === this.tail) {
51          this.tail.next = node;
52          this.tail = node;
53        } else {
54          node.next = current.next;
55          current.next = node;
56        }
57        this.numberOfValues++;
58      }
59      current = current.next;
60    }
61  }
62 }
```

```
61 }
62
63 traverse(fn) {
64     let current = this.head;
65     while(current) {
66         if(fn) {
67             fn(current);
68         }
69         current = current.next;
70     }
71 }
72
73 length() {
74     return this.numberOfValues;
75 }
76
77 print() {
78     let string = '';
79     let current = this.head;
80     while(current) {
81         string += `${current.data} `;
82         current = current.next;
83     }
84     console.log(string.trim());
85 }
86 }
87
88 const singlyLinkedList = new SinglyLinkedList();
89 singlyLinkedList.print(); // => ''
90 singlyLinkedList.add(1);
91 singlyLinkedList.add(2);
92 singlyLinkedList.add(3);
93 singlyLinkedList.add(4);
94 singlyLinkedList.print(); // => 1 2 3 4
95 console.log('length is 4:', singlyLinkedList.length()); // => 4
96 singlyLinkedList.remove(3); // remove value
97 singlyLinkedList.print(); // => 1 2 4
98 singlyLinkedList.remove(9); // remove non existing value
99 singlyLinkedList.print(); // => 1 2 4
100 singlyLinkedList.remove(1); // remove head
101 singlyLinkedList.print(); // => 2 4
102 singlyLinkedList.remove(4); // remove tail
103 singlyLinkedList.print(); // => 2
104 console.log('length is 1:', singlyLinkedList.length()); // => 1
105 singlyLinkedList.add(6);
106 singlyLinkedList.print(); // => 2 6
107 singlyLinkedList.insertAfter(3, 2);
108 singlyLinkedList.print(); // => 2 3 6
109 singlyLinkedList.insertAfter(4, 3);
110 singlyLinkedList.print(); // => 2 3 4 6
111 singlyLinkedList.insertAfter(5, 9); // insertAfter a non existing node
112 singlyLinkedList.print(); // => 2 3 4 6
113 singlyLinkedList.insertAfter(5, 4);
114 singlyLinkedList.insertAfter(7, 6); // insertAfter the tail
115 singlyLinkedList.print(); // => 2 3 4 5 6 7
116 singlyLinkedList.add(8); // add node with normal method
117 singlyLinkedList.print(); // => 2 3 4 5 6 7 8
118 console.log('length is 7:', singlyLinkedList.length()); // => 7
119 singlyLinkedList.traverse(node => { node.data = node.data + 10; });
120 singlyLinkedList.print(); // => 12 13 14 15 16 17 18
```

```
121 singlyLinkedList.traverse(node => { console.log(node.data); }); // => 12 13 14 15 16  
    17 18  
122 console.log('length is 7:', singlyLinkedList.length()); // => 7  
123
```

```
1 class Stack {
2   constructor() {
3     this.stack = [];
4   }
5
6   push(value) {
7     this.stack.push(value);
8   }
9
10  pop() {
11    return this.stack.pop();
12  }
13
14  peek() {
15    return this.stack[this.stack.length - 1];
16  }
17
18  length() {
19    return this.stack.length;
20  }
21
22  print() {
23    console.log(this.stack.join(' '));
24  }
25 }
26
27 const stack = new Stack();
28 stack.push(1);
29 stack.push(2);
30 stack.push(3);
31 stack.print(); // => 1 2 3
32 console.log('length is 3:', stack.length()); // => 3
33 console.log('peek is 3:', stack.peek()); // => 3
34 console.log('pop is 3:', stack.pop()); // => 3
35 stack.print(); // => 1 2
36 console.log('pop is 2:', stack.pop()); // => 2
37 console.log('length is 1:', stack.length()); // => 1
38 console.log('pop is 1:', stack.pop()); // => 1
39 stack.print(); // => ''
40 console.log('peek is undefined:', stack.peek()); // => undefined
41 console.log('pop is undefined:', stack.pop()); // => undefined
42
```

```
1 function Node(data) {
2   this.data = data;
3   this.children = [];
4 }
5
6 class Tree {
7   constructor() {
8     this.root = null;
9   }
10
11   add(data, toNodeData) {
12     const node = new Node(data);
13     const parent = toNodeData ? this.findBFS(toNodeData) : null;
14     if(parent) {
15       parent.children.push(node);
16     } else {
17       if(!this.root) {
18         this.root = node;
19       } else {
20         return 'Root node is already assigned';
21       }
22     }
23   }
24
25   remove(data) {
26     if(this.root.data === data) {
27       this.root = null;
28     }
29
30     const queue = [this.root];
31     while(queue.length) {
32       const node = queue.shift();
33       for (let [index, child] of node.children.entries()) {
34         if(child.data === data) {
35           node.children.splice(index, 1);
36         } else {
37           queue.push(child);
38         }
39       }
40     }
41   }
42
43   contains(data) {
44     return !!this.findBFS(data);
45   }
46
47   findBFS(data) {
48     const queue = [this.root];
49     while(queue.length) {
50       const node = queue.shift();
51       if(node.data === data) {
52         return node;
53       }
54       for(const child of node.children) {
55         queue.push(child);
56       }
57     }
58     return null;
59   }
60 }
```

```
61 _preOrder(node, fn) {
62   if(node) {
63     if(fn) {
64       fn(node);
65     }
66     for(const child of node.children) {
67       this._preOrder(child, fn);
68     }
69   }
70 }
71
72 _postOrder(node, fn) {
73   if(node) {
74     for(const child of node.children) {
75       this._postOrder(child, fn);
76     }
77     if(fn) {
78       fn(node);
79     }
80   }
81 }
82
83 traverseDFS(fn, method) {
84   const current = this.root;
85   if(method) {
86     this[`_${method}`](current, fn);
87   } else {
88     this._preOrder(current, fn);
89   }
90 }
91
92 traverseBFS(fn) {
93   const queue = [this.root];
94   while(queue.length) {
95     const node = queue.shift();
96     if(fn) {
97       fn(node);
98     }
99     for(const child of node.children) {
100       queue.push(child);
101     }
102   }
103 }
104
105 print() {
106   if(!this.root) {
107     return console.log('No root node found');
108   }
109   const newline = new Node('|');
110   const queue = [this.root, newline];
111   let string = '';
112   while(queue.length) {
113     const node = queue.shift();
114     string += `${node.data.toString()} `;
115     if(node === newline && queue.length) {
116       queue.push(newline);
117     }
118     for(const child of node.children) {
119       queue.push(child);
120     }
121   }
122 }
```



```
121     }
122     console.log(string.slice(0, -2).trim());
123 }
124
125 printByLevel() {
126     if(!this.root) {
127         return console.log('No root node found');
128     }
129     const newline = new Node('\n');
130     const queue = [this.root, newline];
131     let string = '';
132     while(queue.length) {
133         const node = queue.shift();
134         string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
135         if(node === newline && queue.length) {
136             queue.push(newline);
137         }
138         for(const child of node.children) {
139             queue.push(child);
140         }
141     }
142     console.log(string.trim());
143 }
144 }
145
146 const tree = new Tree();
147 tree.add('ceo');
148 tree.add('cto', 'ceo');
149 tree.add('dev1', 'cto');
150 tree.add('dev2', 'cto');
151 tree.add('dev3', 'cto');
152 tree.add('cfo', 'ceo');
153 tree.add('accountant', 'cfo');
154 tree.add('cmo', 'ceo');
155 tree.print(); // => ceo | cto cfo cmo | dev1 dev2 dev3 accountant
156 tree.printByLevel(); // => ceo \n cto cfo cmo \n dev1 dev2 dev3 accountant
157 console.log('tree contains dev1 is true:', tree.contains('dev1')); // => true
158 console.log('tree contains dev4 is false:', tree.contains('dev4')); // => false
159 console.log('--- BFS');
160 tree.traverseBFS(node => { console.log(node.data); }); // => ceo cto cfo cmo dev1
    dev2 dev3 accountant
161 console.log('--- DFS preOrder');
162 tree.traverseDFS(node => { console.log(node.data); }, 'preOrder'); // => ceo cto dev1
    dev2 dev3 cfo accountant cmo
163 console.log('--- DFS postOrder');
164 tree.traverseDFS(node => { console.log(node.data); }, 'postOrder'); // => dev1 dev2
    dev3 cto accountant cfo cmo ceo
165 tree.remove('cmo');
166 tree.print(); // => ceo | cto cfo | dev1 dev2 dev3 accountant
167 tree.remove('cfo');
168 tree.print(); // => ceo | cto | dev1 dev2 dev3
169
```

```
1 function Node(data) {
2   this.data = data;
3   this.isWord = false;
4   this.prefixes = 0;
5   this.children = {};
6 }
7
8 class Trie {
9   constructor() {
10     this.root = new Node('');
11   }
12
13   add(word) {
14     if(!this.root) {
15       return null;
16     }
17     this._addNode(this.root, word);
18   }
19
20   _addNode(node, word) {
21     if(!node || !word) {
22       return null;
23     }
24     node.prefixes++;
25     const letter = word.charAt(0);
26     let child = node.children[letter];
27     if(!child) {
28       child = new Node(letter);
29       node.children[letter] = child;
30     }
31     const remainder = word.substring(1);
32     if(!remainder) {
33       child.isWord = true;
34     }
35     this._addNode(child, remainder);
36   }
37
38   remove(word) {
39     if(!this.root) {
40       return;
41     }
42     if(this.contains(word)) {
43       this._removeNode(this.root, word);
44     }
45   }
46
47   _removeNode(node, word) {
48     if(!node || !word) {
49       return;
50     }
51     node.prefixes--;
52     const letter = word.charAt(0);
53
54     const child = node.children[letter];
55     if(child) {
56       const remainder = word.substring(1);
57       if(remainder) {
58         if(child.prefixes === 1) {
59           delete node.children[letter];
60         } else {
```

```
61         this._removeNode(child, remainder);
62     }
63 } else {
64     if(child.prefixes === 0) {
65         delete node.children[letter];
66     } else {
67         child.isWord = false;
68     }
69 }
70 }
71 }
72
73 contains(word) {
74     if(!this.root) {
75         return false;
76     }
77     return this._contains(this.root, word);
78 }
79
80 _contains(node, word) {
81     if(!node || !word) {
82         return false;
83     }
84     const letter = word.charAt(0);
85     const child = node.children[letter];
86     if(child) {
87         const remainder = word.substring(1);
88         if(!remainder && child.isWord) {
89             return true;
90         } else {
91             return this._contains(child, remainder);
92         }
93     } else {
94         return false;
95     }
96 }
97
98 countWords() {
99     if(!this.root) {
100         return console.log('No root node found');
101     }
102     const queue = [this.root];
103     let counter = 0;
104     while(queue.length) {
105         const node = queue.shift();
106         if(node.isWord) {
107             counter++;
108         }
109         for(const child in node.children) {
110             if(node.children.hasOwnProperty(child)) {
111                 queue.push(node.children[child]);
112             }
113         }
114     }
115     return counter;
116 }
117
118 getWords() {
119     const words = [];
120     const word = '';
```

```
121     this._getWords(this.root, words, word);
122     return words;
123 }
124
125 _getWords(node, words, word) {
126     for(const child in node.children) {
127         if(node.children.hasOwnProperty(child)) {
128             word += child;
129             if (node.children[child].isWord) {
130                 words.push(word);
131             }
132             this._getWords(node.children[child], words, word);
133             word = word.substring(0, word.length - 1);
134         }
135     }
136 }
137
138 print() {
139     if(!this.root) {
140         return console.log('No root node found');
141     }
142     const newline = new Node('|');
143     const queue = [this.root, newline];
144     let string = '';
145     while(queue.length) {
146         const node = queue.shift();
147         string += `${node.data.toString()} `;
148         if(node === newline && queue.length) {
149             queue.push(newline);
150         }
151         for(const child in node.children) {
152             if(node.children.hasOwnProperty(child)) {
153                 queue.push(node.children[child]);
154             }
155         }
156     }
157     console.log(string.slice(0, -2).trim());
158 }
159
160 printByLevel() {
161     if(!this.root) {
162         return console.log('No root node found');
163     }
164     const newline = new Node('\n');
165     const queue = [this.root, newline];
166     let string = '';
167     while(queue.length) {
168         const node = queue.shift();
169         string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
170         if(node === newline && queue.length) {
171             queue.push(newline);
172         }
173         for(const child in node.children) {
174             if(node.children.hasOwnProperty(child)) {
175                 queue.push(node.children[child]);
176             }
177         }
178     }
179     console.log(string.trim());
180 }
```

```
181 }
182
183 const trie = new Trie();
184 trie.add('one');
185 trie.add('two');
186 trie.add('fifth');
187 trie.add('fifty');
188 trie.print(); // => | o t f | n w i | e o f | t | h y
189 trie.printByLevel(); // => o t f \n n w i \n e o f \n t \n h y
190 console.log('words are: one, two, fifth, fifty:', trie.getWords()); // => [ 'one',
    'two', 'fifth', 'fifty' ]
191 console.log('trie count words is 4:', trie.countWords()); // => 4
192 console.log('trie contains one is true:', trie.contains('one')); // => true
193 console.log('trie contains on is false:', trie.contains('on')); // => false
194 trie.remove('one');
195 console.log('trie contains one is false:', trie.contains('one')); // => false
196 console.log('trie count words is 3:', trie.countWords()); // => 3
197 console.log('words are two, fifth, fifty:', trie.getWords()); // => [ 'two', 'fifth',
    'fifty' ]
198
```