

```

1 //-----
2 //-----
3 //-----
4
5
6
7 class ArrayADT {
8     constructor() {
9         this.array = [];
10    }
11
12    add( data ) {
13        this.array.push( data );
14    }
15
16    remove( data ) {
17        this.array = this.array.filter( ( current ) => current !== data );
18    }
19
20    search( data ) {
21        const foundIndex = this.array.indexOf( data );
22        if ( foundIndex === -1 ) {
23            return foundIndex;
24        }
25
26        return null;
27    }
28
29    getAtIndex( index ) {
30        return this.array[ index ];
31    }
32
33    length() {
34        return this.array.length;
35    }
36
37    print() {
38        console.log( this.array.join( ' ' ) );
39    }
40 }
41
42 const array = new ArrayADT();
43 console.log( 'const array = new ArrayADT();: ', array );
44 console.log( '-----' );
45
46 console.log( 'array.add(1): ', array.add( 1 ) );
47 array.add( 3 );
48 array.add( 4 );
49 console.log(
50     'array.add(2): ',
51     array.add( 2 ),
52     'array.add(3): ',
53     array.add( 3 ),
54     'array.add(4): ',
55     array.add( 4 )
56 );
57
58 console.log( '-----' );
59 array.print();
60 console.log( '-----' );
61
62 console.log( 'search 3 gives index 2:', array.search( 3 ) );
63 console.log( '-----' );
64
65 console.log( 'getAtIndex 2 gives 3:', array.getAtIndex( 2 ) );
66 console.log( '-----' );
67
68 console.log( 'length is 4:', array.length() );
69 console.log( '-----' );
70
71 array.remove( 3 );
72 array.print();
73 console.log( '-----' );
74

```

```

75 array.add( 5 );
76 array.add( 5 );
77 array.print();
78 console.log( '-----' );
79
80 array.remove( 5 );
81 array.print();
82 console.log( '-----' );
83 /*
84 ~ final : (master) node 01-array.js
85 const array = new ArrayADT(); ArrayADT { array: [] }
86 -----
87 array.add(1): undefined
88 array.add(2);: undefined array.add(3); undefined array.add(4); undefined
89 -----
90 1 3 4 2 3 4
91 -----
92 search 3 gives index 2: null
93 -----
94 getAtIndex 2 gives 3: 4
95 -----
96 length is 4: 6
97 -----
98 1 4 2 4
99 -----
100 1 4 2 4 5 5
101 -----
102 1 4 2 4
103 -----
104 ~ final : (master)
105 */
106 //-----
107 //-----
108 //-----
109 // 1. Creating Arrays
110 let firstArray = [ "a", "b", "c" ];
111 let secondArray = [ "d", "e", "f" ];
112
113 // 2. Access an Array Item
114 console.log( firstArray[ 0 ] ); // Results: "a"
115
116 // 3. Loop over an Array
117 firstArray.forEach( (item, index, array) => {
118     console.log( item, index );
119 } );
120 // Results:
121 // a 0
122 // b 1
123 // c 2
124
125 // 4. Add new item to END of array
126 secondArray.push( 'g' );
127 console.log( secondArray );
128 // Results: ["d","e","f", "g"]
129
130 // 5. Remove item from END of array
131 secondArray.pop();
132 console.log( secondArray );
133 // Results: ["d","e","f"]
134
135 // 6. Remove item from FRONT of array
136 secondArray.shift();
137 console.log( secondArray );
138 // Results: ["e","f"]
139
140 // 7. Add item to FRONT of array
141 secondArray.unshift( "d" );
142 console.log( secondArray );
143 // Results: ["d","e","f"]
144
145 // 8. Find INDEX of an item in array
146 let position = secondArray.indexOf( 'f' );
147 // Results: 2
148

```

```

149 // 9. Remove Item by Index Position
150 secondArray.splice( position, 1 );
151 console.log( secondArray );
152 // Note, the second argument, in this case "1",
153 // represent the number of array elements to be removed
154 // Results:  ["d","e"]
155
156 // 10. Copy an Array
157 let shallowCopy = secondArray.slice();
158 console.log( secondArray );
159 console.log( shallowCopy );
160 // Results: shallowCopy === ["d","e"]
161
162 // 11. JavaScript properties that BEGIN with a digit MUST be accessed using bracket notation
163 renderer [ '.3d' ].setTexture( model, 'character.png' ); // a syntax error
164 renderer[ '3d' ].setTexture( model, 'character.png' ); // works properly
165
166
167 // 12. Combine two Arrays
168 let thirdArray = firstArray.concat( secondArray );
169 console.log( thirdArray );
170 // ["a","b","c", "d", "e"];
171
172 // 13. Combine all Array elements into a string
173 console.log( thirdArray.join() ); // Results: a,b,c,d,e
174 console.log( thirdArray.join( ' ' ) ); // Results: abcde
175 console.log( thirdArray.join( '-' ) ); // Results: a-b-c-d-e
176
177 // 14. Reversing an Array (in place, i.e. destructive)
178 console.log( thirdArray.reverse() ); // ["e", "d", "c", "b", "a"]
179
180 // 15. sort
181 let unsortedArray = [ "Alphabet", "Zoo", "Products", "Computer Science", "Computer" ];
182 console.log( unsortedArray.sort() );
183 // Results: ["Alphabet", "Computer", "Computer Science", "Products", "Zoo" ]
184
185 //-----
186 //-----
187 //-----
188
189
190 // 16. Creating an Object
191
192 let newObj = {
193     name: "I'm an object",
194     values: [ 1, 10, 11, 20 ],
195     others: '',
196     '1property': 'example of property name starting with digit',
197 };
198
199 // 17. Figure out what keys/properties are in an object
200 console.log( Object.keys( newObj ) );
201 // Results: [ 'name', 'values', 'others', '1property' ]
202
203 // 18. Show all values stored in the object
204 console.log( Object.values( newObj ) );
205
206 // Results:
207 // [ 'I\'m an object',
208 //   [ 1, 10, 11, 20 ],
209 //   '',
210 //   'example of property name starting with digit' ]
211
212 // 19. Show all key and values of the object
213 for ( let [ key, value ] of Object.entries( newObj ) ) {
214     console.log( `${key}: ${value}` );
215 }
216 // Results:
217 // name: I'm an object
218 // values: 1,10,11,20
219 // others:
220 // 1property: example of property name starting with digit
221
222 // 20. Accessing Object's Properties

```

```

223 // Two different ways to access properties, both produce same results
224 console.log( newObj.name );
225 console.log( newObj[ 'name' ] );
226
227 // But if the property name starts with a digit,
228 // we CANNOT use dot notation
229 console.log( newObj[ '1property' ] );
230
231 // 21. Adding a Method to an Object
232 newObj.helloWorld = () => {
233     console.log( 'Hello World from inside an object!' );
234 };
235
236 // 22. Invoking an Object's Method
237 newObj.helloWorld();
238 //-----
239 //-----
240 //-----
241 class HashTable {
242     constructor( size ) {
243         this.values = {};
244         this.numberOfValues = 0;
245         this.size = size;
246     }
247     add( key, value ) {
248         let hash = this.calculateHash( key );
249         if ( !this.values.hasOwnProperty( hash ) ) {
250             this.values[ hash ] = {};
251         }
252         if ( !this.values[ hash ].hasOwnProperty( key ) ) {
253             this.numberOfValues++;
254         }
255         this.values[ hash ][ key ] = value;
256     }
257     remove( key ) {
258         let hash = this.calculateHash( key );
259         if (
260             this.values.hasOwnProperty( hash ) &&
261             this.values[ hash ].hasOwnProperty( key )
262         ) {
263             delete this.values[ hash ][ key ];
264             this.numberOfValues--;
265         }
266     }
267     calculateHash( key ) {
268         return key.toString().length % this.size;
269     }
270     search( key ) {
271         let hash = this.calculateHash( key );
272         if (
273             this.values.hasOwnProperty( hash ) &&
274             this.values[ hash ].hasOwnProperty( key )
275         ) {
276             return this.values[ hash ][ key ];
277         } else {
278             return null;
279         }
280     }
281     length() {
282         return this.numberOfValues;
283     }
284     print() {
285         let string = '';
286         for ( let value in this.values ) {
287             for ( let key in this.values[ value ] ) {
288                 string += this.values[ value ][ key ] + ' ';
289             }
290         }
291         console.log( string.trim() );
292     }
293 }
294 let hashTable = new HashTable( 3 );
295 hashTable.add( 'first', 1 );
296 hashTable.add( 'second', 2 );

```

```

297 hashTable.add( 'third', 3 );
298 hashTable.add( 'fourth', 4 );
299 hashTable.add( 'fifth', 5 );
300 hashTable.print(); // => 2 4 1 3 5
301 console.log( 'length gives 5:', hashTable.length() ); // => 5
302 console.log( 'search second gives 2:', hashTable.search( 'second' ) ); // => 2
303 hashTable.remove( 'fourth' );
304 hashTable.remove( 'first' );
305 hashTable.print(); // => 2 3 5
306 console.log( 'length gives 3:', hashTable.length() ); // => 3
307 /*
308     ~ js-files : (master) node hash.js
309     2 4 1 3 5
310     length gives 5: 5
311     search second gives 2: 2
312     2 3 5
313     length gives 3: 3
314 */
315 //-----
316 //-----
317 //-----
318 // 23. Creating a new Set
319 let newSet = new Set();
320
321 // 24. Adding new elements to a set
322 newSet.add( 1 ); // Set[1]
323 newSet.add( 'text' ); // Set[1, "text"]
324
325 // 25. Check if element is in set
326 newSet.has( 1 ); // true
327
328 // 24. Check size of set
329 console.log( newSet.size ); // Results: 2
330
331 // 26. Delete element from set
332 newSet.delete( 1 ); // Set["text"]
333
334 // 27. Set Operations: isSuperset
335 function isSuperset( set, subset ) {
336     for ( let elem of subset ) {
337         if ( !set.has( elem ) ) {
338             return false;
339         }
340     }
341     return true;
342 }
343
344 // 28. Set Operations: union
345 function union( setA, setB ) {
346     let _union = new Set( setA );
347     for ( let elem of setB ) {
348         _union.add( elem );
349     }
350     return _union;
351 }
352
353 // 29. Set Operations: intersection
354 function intersection( setA, setB ) {
355     let _intersection = new Set();
356     for ( let elem of setB ) {
357         if ( setA.has( elem ) ) {
358             _intersection.add( elem );
359         }
360     }
361     return _intersection;
362 }
363
364 // 30. Set Operations: symmetricDifference
365 function symmetricDifference( setA, setB ) {
366     let _difference = new Set( setA );
367     for ( let elem of setB ) {
368         if ( _difference.has( elem ) ) {
369             _difference.delete( elem );
370         } else {
371             _difference.add( elem );
372         }
373     }
374 }

```

```

371 }
372 return _difference;
373 }
374 // 31. Set Operations: difference
375 function difference( setA, setB ) {
376     let _difference = new Set( setA );
377     for ( let elem of setB ) {
378         _difference.delete( elem );
379     }
380     return _difference;
381 }
382
383 // Examples
384 let setA = new Set( [ 1, 2, 3, 4 ] );
385 let setB = new Set( [ 2, 3 ] );
386 let setC = new Set( [ 3, 4, 5, 6 ] );
387
388 console.log( isSuperset( setA, setB ) ); // => true
389 console.log( union( setA, setC ) ); // => Set [1, 2, 3, 4, 5, 6]
390 console.log( intersection( setA, setC ) ); // => Set [3, 4]
391 console.log( symmetricDifference( setA, setC ) ); // => Set [1, 2, 5, 6]
392 console.log( difference( setA, setC ) ); // => Set [1, 2]
393
394 class Set {
395     constructor() {
396         this.values = [];
397         this.numberOfValues = 0;
398     }
399
400     add(value) {
401         if ( !~this.values.indexOf( value ) ) {
402             this.values.push( value );
403             this.numberOfValues++;
404         }
405     }
406
407     remove(value) {
408         let index = this.values.indexOf( value );
409         if ( ~index ) {
410             this.values.splice( index, 1 );
411             this.numberOfValues--;
412         }
413     }
414
415     contains(value) {
416         return this.values.indexOf( value ) !== -1;
417     }
418
419     union(set) {
420         let newSet = new Set();
421         set.values.forEach( value => {
422             newSet.add( value );
423         } );
424         this.values.forEach( value => {
425             newSet.add( value );
426         } );
427         return newSet;
428     }
429
430     intersect(set) {
431         let newSet = new Set();
432         this.values.forEach( value => {
433             if ( set.contains( value ) ) {
434                 newSet.add( value );
435             }
436         } );
437         return newSet;
438     }
439
440     difference(set) {
441         let newSet = new Set();
442         this.values.forEach( value => {
443             if ( !set.contains( value ) ) {
444                 newSet.add( value );

```

```

445     }
446   } );
447   return newSet;
448 }
449
450 isSubset(set) {
451   return set.values.every( function ( value ) {
452     return this.contains( value );
453   }, this );
454 }
455
456 length() {
457   return this.numberOfValues;
458 }
459
460 print() {
461   console.log( this.values.join( ' ' ) );
462 }
463 }
464
465 let set = new Set();
466 set.add( 1 );
467 set.add( 2 );
468 set.add( 3 );
469 set.add( 4 );
470 set.print(); // => 1 2 3 4
471 set.remove( 3 );
472 set.print(); // => 1 2 4
473 console.log( 'contains 4 is true:', set.contains( 4 ) ); // => true
474 console.log( 'contains 3 is false:', set.contains( 3 ) ); // => false
475 console.log( '---' );
476 let set1 = new Set();
477 set1.add( 1 );
478 set1.add( 2 );
479 let set2 = new Set();
480 set2.add( 2 );
481 set2.add( 3 );
482 let set3 = set2.union( set1 );
483 set3.print(); // => 1 2 3
484 let set4 = set2.intersect( set1 );
485 set4.print(); // => 2
486 let set5 = set.difference( set3 ); // 1 2 4 diff 1 2 3
487 set5.print(); // => 4
488 let set6 = set3.difference( set ); // 1 2 3 diff 1 2 4
489 set6.print(); // => 3
490 console.log( 'set1 subset of set is true:', set.isSubset( set1 ) ); // => true
491 console.log( 'set2 subset of set is false:', set.isSubset( set2 ) ); // => false
492 console.log( 'set1 length gives 2:', set1.length() ); // => 2
493 console.log( 'set3 length gives 3:', set3.length() ); // => 3
494 //-----
495 //-----
496 //-----
497 function Node( data ) {
498   this.data = data;
499   this.next = null;
500 }
501
502 class SinglyLinkedList {
503   constructor() {
504     this.head = null;
505     this.tail = null;
506     this.numberOfValues = 0;
507   }
508
509   add(data) {
510     let node = new Node( data );
511     if ( !this.head ) {
512       this.head = node;
513       this.tail = node;
514     } else {
515       this.tail.next = node;
516       this.tail = node;
517     }
518     this.numberOfValues++;

```

```

519 }
520
521 remove(data) {
522     let previous = this.head;
523     let current = this.head;
524     while ( current ) {
525         if ( current.data === data ) {
526             if ( current === this.head ) {
527                 this.head = this.head.next;
528             }
529             if ( current === this.tail ) {
530                 this.tail = previous;
531             }
532             previous.next = current.next;
533             this.numberOfValues--;
534         } else {
535             previous = current;
536         }
537         current = current.next;
538     }
539 }
540
541 insertAfter(data, toNodeData) {
542     let current = this.head;
543     while ( current ) {
544         if ( current.data === toNodeData ) {
545             let node = new Node( data );
546             if ( current === this.tail ) {
547                 this.tail.next = node;
548                 this.tail = node;
549             } else {
550                 node.next = current.next;
551                 current.next = node;
552             }
553             this.numberOfValues++;
554         }
555         current = current.next;
556     }
557 }
558
559 traverse(fn) {
560     let current = this.head;
561     while ( current ) {
562         if ( fn ) {
563             fn( current );
564         }
565         current = current.next;
566     }
567 }
568
569 length() {
570     return this.numberOfValues;
571 }
572
573 print() {
574     let string = '';
575     let current = this.head;
576     while ( current ) {
577         string += current.data + ' ';
578         current = current.next;
579     }
580     console.log( string.trim() );
581 }
582 }
583
584 let singlyLinkedList = new SinglyLinkedList();
585 singlyLinkedList.print(); // => ''
586 singlyLinkedList.add( 1 );
587 singlyLinkedList.add( 2 );
588 singlyLinkedList.add( 3 );
589 singlyLinkedList.add( 4 );
590 singlyLinkedList.print(); // => 1 2 3 4
591 console.log( 'length is 4:', singlyLinkedList.length() ); // => 4
592 singlyLinkedList.remove( 3 ); // remove value

```



```

593 singlyLinkedList.print(); // => 1 2 4
594 singlyLinkedList.remove( 9 ); // remove non existing value
595 singlyLinkedList.print(); // => 1 2 4
596 singlyLinkedList.remove( 1 ); // remove head
597 singlyLinkedList.print(); // => 2 4
598 singlyLinkedList.remove( 4 ); // remove tail
599 singlyLinkedList.print(); // => 2
600 console.log( 'length is 1:', singlyLinkedList.length() ); // => 1
601 singlyLinkedList.add( 6 );
602 singlyLinkedList.print(); // => 2 6
603 singlyLinkedList.insertAfter( 3, 2 );
604 singlyLinkedList.print(); // => 2 3 6
605 singlyLinkedList.insertAfter( 4, 3 );
606 singlyLinkedList.print(); // => 2 3 4 6
607 singlyLinkedList.insertAfter( 5, 9 ); // insertAfter a non existing node
608 singlyLinkedList.print(); // => 2 3 4 6
609 singlyLinkedList.insertAfter( 5, 4 );
610 singlyLinkedList.insertAfter( 7, 6 ); // insertAfter the tail
611 singlyLinkedList.print(); // => 2 3 4 5 6 7
612 singlyLinkedList.add( 8 ); // add node with normal method
613 singlyLinkedList.print(); // => 2 3 4 5 6 7 8
614 console.log( 'length is 7:', singlyLinkedList.length() ); // => 7
615 singlyLinkedList.traverse( node => {
616     node.data = node.data + 10;
617 } );
618 singlyLinkedList.print(); // => 12 13 14 15 16 17 18
619 singlyLinkedList.traverse( node => {
620     console.log( node.data );
621 } ); // => 12 13 14 15 16 17 18
622 console.log( 'length is 7:', singlyLinkedList.length() ); // => 7
623
624 //-----
625 //-----
626 //-----
627 class Graph {
628     constructor() {
629         this.vertices = [];
630         this.edges = [];
631         this.numberOfEdges = 0;
632     }
633
634     addVertex(vertex) {
635         this.vertices.push( vertex );
636         this.edges[ vertex ] = [];
637     }
638
639     removeVertex(vertex) {
640         let index = this.vertices.indexOf( vertex );
641         if ( ~index ) {
642             this.vertices.splice( index, 1 );
643         }
644         while ( this.edges[ vertex ].length ) {
645             let adjacentVertex = this.edges[ vertex ].pop();
646             this.removeEdge( adjacentVertex, vertex );
647         }
648     }
649
650     addEdge(vertex1, vertex2) {
651         this.edges[ vertex1 ].push( vertex2 );
652         this.edges[ vertex2 ].push( vertex1 );
653         this.numberOfEdges++;
654     }
655
656     removeEdge(vertex1, vertex2) {
657         let index1 = this.edges[ vertex1 ] ? this.edges[ vertex1 ].indexOf( vertex2 ) : -1;
658         let index2 = this.edges[ vertex2 ] ? this.edges[ vertex2 ].indexOf( vertex1 ) : -1;
659         if ( ~index1 ) {
660             this.edges[ vertex1 ].splice( index1, 1 );
661             this.numberOfEdges--;
662         }
663         if ( ~index2 ) {
664             this.edges[ vertex2 ].splice( index2, 1 );
665         }
666     }

```

```

667
668 size() {
669     return this.vertices.length;
670 }
671
672 relations() {
673     return this.numberOfEdges;
674 }
675
676 traverseDFS(vertex, fn) {
677     if ( !~this.vertices.indexOf( vertex ) ) {
678         return console.log( 'Vertex not found' );
679     }
680     let visited = [];
681     this._traverseDFS( vertex, visited, fn );
682 }
683
684 _traverseDFS(vertex, visited, fn) {
685     visited[ vertex ] = true;
686     if ( this.edges[ vertex ] !== undefined ) {
687         fn( vertex );
688     }
689     for ( let i = 0; i < this.edges[ vertex ].length; i++ ) {
690         if ( !visited[ this.edges[ vertex ][ i ] ] ) {
691             this._traverseDFS( this.edges[ vertex ][ i ], visited, fn );
692         }
693     }
694 }
695
696 traverseBFS(vertex, fn) {
697     if ( !~this.vertices.indexOf( vertex ) ) {
698         return console.log( 'Vertex not found' );
699     }
700     let queue = [];
701     queue.push( vertex );
702     let visited = [];
703     visited[ vertex ] = true;
704
705     while ( queue.length ) {
706         vertex = queue.shift();
707         fn( vertex );
708         for ( let i = 0; i < this.edges[ vertex ].length; i++ ) {
709             if ( !visited[ this.edges[ vertex ][ i ] ] ) {
710                 visited[ this.edges[ vertex ][ i ] ] = true;
711                 queue.push( this.edges[ vertex ][ i ] );
712             }
713         }
714     }
715 }
716
717 pathFromTo(vertexSource, vertexDestination) {
718     if ( !~this.vertices.indexOf( vertexSource ) ) {
719         return console.log( 'Vertex not found' );
720     }
721     let queue = [];
722     queue.push( vertexSource );
723     let visited = [];
724     visited[ vertexSource ] = true;
725     let paths = [];
726
727     while ( queue.length ) {
728         let vertex = queue.shift();
729         for ( let i = 0; i < this.edges[ vertex ].length; i++ ) {
730             if ( !visited[ this.edges[ vertex ][ i ] ] ) {
731                 visited[ this.edges[ vertex ][ i ] ] = true;
732                 queue.push( this.edges[ vertex ][ i ] );
733                 // save paths between vertices
734                 paths[ this.edges[ vertex ][ i ] ] = vertex;
735             }
736         }
737     }
738     if ( !visited[ vertexDestination ] ) {
739         return undefined;
740     }

```

```

741
742     let path = [];
743     for ( let j = vertexDestination; j != vertexSource; j = paths[ j ] ) {
744         path.push( j );
745     }
746     path.push( j );
747     return path.reverse().join( '-' );
748 }
749
750 print() {
751     console.log( this.vertices.map( function ( vertex ) {
752         return ( vertex + ' -> ' + this.edges[ vertex ].join( ', ' ) ).trim();
753     }, this ).join( ' | ' ) );
754 }
755 }
756
757 let graph = new Graph();
758 graph.addVertex( 1 );
759 graph.addVertex( 2 );
760 graph.addVertex( 3 );
761 graph.addVertex( 4 );
762 graph.addVertex( 5 );
763 graph.addVertex( 6 );
764 graph.print(); // 1 -> | 2 -> | 3 -> | 4 -> | 5 -> | 6 ->
765 graph.addEdge( 1, 2 );
766 graph.addEdge( 1, 5 );
767 graph.addEdge( 2, 3 );
768 graph.addEdge( 2, 5 );
769 graph.addEdge( 3, 4 );
770 graph.addEdge( 4, 5 );
771 graph.addEdge( 4, 6 );
772 graph.print(); // 1 -> 2, 5 | 2 -> 1, 3, 5 | 3 -> 2, 4 | 4 -> 3, 5, 6 | 5 -> 1, 2, 4 | 6 -> 4
773 console.log( 'graph size (number of vertices):', graph.size() ); // => 6
774 console.log( 'graph relations (number of edges):', graph.relations() ); // => 7
775 graph.traverseDFS( 1, vertex => {
776     console.log( vertex );
777 } ); // => 1 2 3 4 5 6
778 console.log( '---' );
779 graph.traverseBFS( 1, vertex => {
780     console.log( vertex );
781 } ); // => 1 2 5 3 4 6
782 graph.traverseDFS( 0, vertex => {
783     console.log( vertex );
784 } ); // => 'Vertex not found'
785 graph.traverseBFS( 0, vertex => {
786     console.log( vertex );
787 } ); // => 'Vertex not found'
788 console.log( 'path from 6 to 1:', graph.pathFromTo( 6, 1 ) ); // => 6-4-5-1
789 console.log( 'path from 3 to 5:', graph.pathFromTo( 3, 5 ) ); // => 3-2-5
790 graph.removeEdge( 1, 2 );
791 graph.removeEdge( 4, 5 );
792 graph.removeEdge( 10, 11 );
793 console.log( 'graph relations (number of edges):', graph.relations() ); // => 5
794 console.log( 'path from 6 to 1:', graph.pathFromTo( 6, 1 ) ); // => 6-4-3-2-5-1
795 graph.addEdge( 1, 2 );
796 graph.addEdge( 4, 5 );
797 console.log( 'graph relations (number of edges):', graph.relations() ); // => 7
798 console.log( 'path from 6 to 1:', graph.pathFromTo( 6, 1 ) ); // => 6-4-5-1
799 graph.removeVertex( 5 );
800 console.log( 'graph size (number of vertices):', graph.size() ); // => 5
801 console.log( 'graph relations (number of edges):', graph.relations() ); // => 4
802 console.log( 'path from 6 to 1:', graph.pathFromTo( 6, 1 ) ); // => 6-4-3-2-1
803 //-----
804 //-----
805 //-----
806

```