```
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------



class ArrayADT {
  constructor() {
    this.array = [];
  }

  add( data ) {
    this.array.push( data );
  }

  remove( data ) {
    this.array = this.array.filter( ( current ) => current !== data );
  }

  search( data ) {
    const foundIndex = this.array.indexOf( data );
    if ( foundIndex === -1 ) {
      return foundIndex;
    }

    return null;
  }

  getAtIndex( index ) {
    return this.array[ index ];
  }

  length() {
    return this.array.length;
  }

  print() {
    console.log( this.array.join( ' ' ) );
  }
}

const array = new ArrayADT();
console.log( 'const array = new ArrayADT();: ', array );
console.log( '------------------------------' );

console.log( 'array.add(1): ', array.add( 1 ) );
array.add( 3 );
array.add( 4 );
console.log(
  'array.add(2);: ',
  array.add( 2 ),
  'array.add(3);',
  array.add( 3 ),
  'array.add(4); ',
  array.add( 4 )
);

console.log( '------------------------------' );
array.print();
console.log( '------------------------------' );

console.log( 'search 3 gives index 2:', array.search( 3 ) );
console.log( '------------------------------' );

console.log( 'getAtIndex 2 gives 3:', array.getAtIndex( 2 ) );
console.log( '------------------------------' );

console.log( 'length is 4:', array.length() );
console.log( '------------------------------' );

array.remove( 3 );
array.print();
console.log( '------------------------------' );

```

```
array.add( 5 );
array.add( 5 );
array.print();
console.log( '------------------------------' );

array.remove( 5 );
array.print();
console.log( '------------------------------' );
/*
     ~ final : (master) node 01-array.js
    const array = new ArrayADT();:  ArrayADT { array: [] }
    --------------------------------
    array.add(1):  undefined
    array.add(2);:  undefined array.add(3); undefined array.add(4);  undefined
    --------------------------------
    1 3 4 2 3 4
    --------------------------------
    search 3 gives index 2: null
    --------------------------------
    getAtIndex 2 gives 3: 4
    --------------------------------
    length is 4: 6
    --------------------------------
    1 4 2 4
    --------------------------------
    1 4 2 4 5 5
    --------------------------------
    1 4 2 4
    --------------------------------
     ~ final : (master)
     */
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// 1. Creating Arrays
let firstArray = [ "a", "b", "c" ];
let secondArray = [ "d", "e", "f" ];

// 2. Access an Array Item
console.log( firstArray[ 0 ] ); // Results: "a"

// 3. Loop over an Array
firstArray.forEach( (item, index, array) => {
  console.log( item, index );
} );
// Results:
// a 0
// b 1
// c 2

// 4. Add new item to END of array
secondArray.push( 'g' );
console.log( secondArray );
// Results: ["d","e","f", "g"]

// 5. Remove item from END of array
secondArray.pop();
console.log( secondArray );
// Results: ["d","e","f"]

// 6. Remove item from FRONT of array
secondArray.shift();
console.log( secondArray );
// Results: ["e","f"]

// 7. Add item to FRONT of array
secondArray.unshift( "d" );
console.log( secondArray );
// Results: ["d","e","f"]

// 8. Find INDEX of an item in array
let position = secondArray.indexOf( 'f' );
// Results: 2

```

```
// 9. Remove Item by Index Position
secondArray.splice( position, 1 );
console.log( secondArray );
// Note, the second argument, in this case "1",
// represent the number of array elements to be removed
// Results:  ["d","e"]

// 10. Copy an Array
let shallowCopy = secondArray.slice();
console.log( secondArray );
console.log( shallowCopy );
// Results: ShallowCopy === ["d","e"]

// 11. JavaScript properties that BEGIN with a digit MUST be accessed using bracket notation
renderer ['.3d'].setTexture( model, 'character.png' ); // a syntax error
renderer[ '3d' ].setTexture( model, 'character.png' ); // works properly


// 12. Combine two Arrays
let thirdArray = firstArray.concat( secondArray );
console.log( thirdArray );
// ["a","b","c", "d", "e"];

// 13. Combine all Array elements into a string
console.log( thirdArray.join() ); // Results: a,b,c,d,e
console.log( thirdArray.join( '' ) ); // Results: abcde
console.log( thirdArray.join( '-' ) ); // Results: a-b-c-d-e

// 14. Reversing an Array (in place, i.e. destructive)
console.log( thirdArray.reverse() ); // ["e", "d", "c", "b", "a"]

// 15. sort
let unsortedArray = [ "Alphabet", "Zoo", "Products", "Computer Science", "Computer" ];
console.log( unsortedArray.sort() );
// Results: ["Alphabet", "Computer", "Computer Science", "Products", "Zoo" ]

//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------


// 16. Creating an Object

let newObj = {
  name: "I'm an object",
  values: [ 1, 10, 11, 20 ],
  others: '',
  '1property': 'example of property name starting with digit',
};

// 17. Figure out what keys/properties are in an object
console.log( Object.keys( newObj ) );
// Results: [ 'name', 'values', 'others', '1property' ]

// 18. Show all values stored in the object
console.log( Object.values( newObj ) );

// Results:
// [ 'I\'m an object',
//   [ 1, 10, 11, 20 ],
//   '',
//   'example of property name starting with digit' ]

// 19. Show all key and values of the object
for ( let [ key, value ] of Object.entries( newObj ) ) {
  console.log( `${key}: ${value}` );
}
// Results:
// name: I'm an object
// values: 1,10,11,20
// others:
// 1property: example of property name starting with digit

// 20. Accessing Object's Properties
```

```
// Two different ways to access properties, both produce same results
console.log( newObj.name );
console.log( newObj[ 'name' ] );

// But if the property name starts with a digit,
// we CANNOT use dot notation
console.log( newObj[ '1property' ] );

// 21. Adding a Method to an Object
newObj.helloWorld = () => {
  console.log( 'Hello World from inside an object!' );
};

// 22. Invoking an Object's Method
newObj.helloWorld();
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
class HashTable {
  constructor( size ) {
    this.values = {};
    this.numberOfValues = 0;
    this.size = size;
  }
  add( key, value ) {
    let hash = this.calculateHash( key );
    if ( !this.values.hasOwnProperty( hash ) ) {
      this.values[ hash ] = {};
    }
    if ( !this.values[ hash ].hasOwnProperty( key ) ) {
      this.numberOfValues++;
    }
    this.values[ hash ][ key ] = value;
  }
  remove( key ) {
    let hash = this.calculateHash( key );
    if (
      this.values.hasOwnProperty( hash ) &&
      this.values[ hash ].hasOwnProperty( key )
    ) {
      delete this.values[ hash ][ key ];
      this.numberOfValues--;
    }
  }
  calculateHash( key ) {
    return key.toString().length % this.size;
  }
  search( key ) {
    let hash = this.calculateHash( key );
    if (
      this.values.hasOwnProperty( hash ) &&
      this.values[ hash ].hasOwnProperty( key )
    ) {
      return this.values[ hash ][ key ];
    } else {
      return null;
    }
  }
  length() {
    return this.numberOfValues;
  }
  print() {
    let string = '';
    for ( let value in this.values ) {
      for ( let key in this.values[ value ] ) {
        string += this.values[ value ][ key ] + ' ';
      }
    }
    console.log( string.trim() );
  }
}
let hashTable = new HashTable( 3 );
hashTable.add( 'first', 1 );
hashTable.add( 'second', 2 );
```

```
hashTable.add( 'third', 3 );
hashTable.add( 'fourth', 4 );
hashTable.add( 'fifth', 5 );
hashTable.print(); // => 2 4 1 3 5
console.log( 'length gives 5:', hashTable.length() ); // => 5
console.log( 'search second gives 2:', hashTable.search( 'second' ) ); // => 2
hashTable.remove( 'fourth' );
hashTable.remove( 'first' );
hashTable.print(); // => 2 3 5
console.log( 'length gives 3:', hashTable.length() ); // => 3
/*
       ~ js-files : (master) node hash.js
    2 4 1 3 5
    length gives 5: 5
    search second gives 2: 2
    2 3 5
    length gives 3: 3
    */
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// 23. Creating a new Set
let newSet = new Set();

// 24. Adding new elements to a set
newSet.add( 1 ); // Set[1]
newSet.add( 'text' ); // Set[1, "text"]

// 25. Check if element is in set
newSet.has( 1 ); // true

// 24. Check size of set
console.log( newSet.size ); // Results: 2

// 26. Delete element from set
newSet.delete( 1 ); // Set["text"]

// 27. Set Operations: isSuperSet
function isSuperset( set, subset ) {
  for ( let elem of subset ) {
    if ( !set.has( elem ) ) {
      return false;
    }
  }
  return true;
}
// 28. Set Operations: union
function union( setA, setB ) {
  let _union = new Set( setA );
  for ( let elem of setB ) {
    _union.add( elem );
  }
  return _union;
}

// 29. Set Operations: intersection
function intersection( setA, setB ) {
  let _intersection = new Set();
  for ( let elem of setB ) {
    if ( setA.has( elem ) ) {
      _intersection.add( elem );
    }
  }
  return _intersection;
}
// 30. Set Operations: symmetricDifference
function symmetricDifference( setA, setB ) {
  let _difference = new Set( setA );
  for ( let elem of setB ) {
    if ( _difference.has( elem ) ) {
      _difference.delete( elem );
    } else {
      _difference.add( elem );
    }
```

```
  }
  return _difference;
}
// 31. Set Operations: difference
function difference( setA, setB ) {
  let _difference = new Set( setA );
  for ( let elem of setB ) {
    _difference.delete( elem );
  }
  return _difference;
}

// Examples
let setA = new Set( [ 1, 2, 3, 4 ] );
let setB = new Set( [ 2, 3 ] );
let setC = new Set( [ 3, 4, 5, 6 ] );

console.log( isSuperset( setA, setB ) ); // => true
console.log( union( setA, setC ) ); // => Set [1, 2, 3, 4, 5, 6]
console.log( intersection( setA, setC ) ); // => Set [3, 4]
console.log( symmetricDifference( setA, setC ) ); // => Set [1, 2, 5, 6]
console.log( difference( setA, setC ) ); // => Set [1, 2]

class Set {
  constructor() {
    this.values = [];
    this.numberOfValues = 0;
  }

  add(value) {
    if ( !~this.values.indexOf( value ) ) {
      this.values.push( value );
      this.numberOfValues++;
    }
  }

  remove(value) {
    let index = this.values.indexOf( value );
    if ( ~index ) {
      this.values.splice( index, 1 );
      this.numberOfValues--;
    }
  }

  contains(value) {
    return this.values.indexOf( value ) !== -1;
  }

  union(set) {
    let newSet = new Set();
    set.values.forEach( value => {
      newSet.add( value );
    } );
    this.values.forEach( value => {
      newSet.add( value );
    } );
    return newSet;
  }

  intersect(set) {
    let newSet = new Set();
    this.values.forEach( value => {
      if ( set.contains( value ) ) {
        newSet.add( value );
      }
    } );
    return newSet;
  }

  difference(set) {
    let newSet = new Set();
    this.values.forEach( value => {
      if ( !set.contains( value ) ) {
        newSet.add( value );
```

```
      }
    } );
    return newSet;
  }

  isSubset(set) {
    return set.values.every( function ( value ) {
      return this.contains( value );
    }, this );
  }

  length() {
    return this.numberOfValues;
  }

  print() {
    console.log( this.values.join( ' ' ) );
  }
}

let set = new Set();
set.add( 1 );
set.add( 2 );
set.add( 3 );
set.add( 4 );
set.print(); // => 1 2 3 4
set.remove( 3 );
set.print(); // => 1 2 4
console.log( 'contains 4 is true:', set.contains( 4 ) ); // => true
console.log( 'contains 3 is false:', set.contains( 3 ) ); // => false
console.log( '---' );
let set1 = new Set();
set1.add( 1 );
set1.add( 2 );
let set2 = new Set();
set2.add( 2 );
set2.add( 3 );
let set3 = set2.union( set1 );
set3.print(); // => 1 2 3
let set4 = set2.intersect( set1 );
set4.print(); // => 2
let set5 = set.difference( set3 ); // 1 2 4 diff 1 2 3
set5.print(); // => 4
let set6 = set3.difference( set ); // 1 2 3 diff 1 2 4
set6.print(); // => 3
console.log( 'set1 subset of set is true:', set.isSubset( set1 ) ); // => true
console.log( 'set2 subset of set is false:', set.isSubset( set2 ) ); // => false
console.log( 'set1 length gives 2:', set1.length() ); // => 2
console.log( 'set3 length gives 3:', set3.length() ); // => 3
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
function Node( data ) {
  this.data = data;
  this.next = null;
}

class SinglyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
    this.numberOfValues = 0;
  }

  add(data) {
    let node = new Node( data );
    if ( !this.head ) {
      this.head = node;
      this.tail = node;
    } else {
      this.tail.next = node;
      this.tail = node;
    }
    this.numberOfValues++;
```

```
  }

  remove(data) {
    let previous = this.head;
    let current = this.head;
    while ( current ) {
      if ( current.data === data ) {
        if ( current === this.head ) {
          this.head = this.head.next;
        }
        if ( current === this.tail ) {
          this.tail = previous;
        }
        previous.next = current.next;
        this.numberOfValues--;
      } else {
        previous = current;
      }
      current = current.next;
    }
  }

  insertAfter(data, toNodeData) {
    let current = this.head;
    while ( current ) {
      if ( current.data === toNodeData ) {
        let node = new Node( data );
        if ( current === this.tail ) {
          this.tail.next = node;
          this.tail = node;
        } else {
          node.next = current.next;
          current.next = node;
        }
        this.numberOfValues++;
      }
      current = current.next;
    }
  }

  traverse(fn) {
    let current = this.head;
    while ( current ) {
      if ( fn ) {
        fn( current );
      }
      current = current.next;
    }
  }

  length() {
    return this.numberOfValues;
  }

  print() {
    let string = '';
    let current = this.head;
    while ( current ) {
      string += current.data + ' ';
      current = current.next;
    }
    console.log( string.trim() );
  }
}

let singlyLinkedList = new SinglyLinkedList();
singlyLinkedList.print(); // => ''
singlyLinkedList.add( 1 );
singlyLinkedList.add( 2 );
singlyLinkedList.add( 3 );
singlyLinkedList.add( 4 );
singlyLinkedList.print(); // => 1 2 3 4
console.log( 'length is 4:', singlyLinkedList.length() ); // => 4
singlyLinkedList.remove( 3 ); // remove value
```

```
singlyLinkedList.print(); // => 1 2 4
singlyLinkedList.remove( 9 ); // remove non existing value
singlyLinkedList.print(); // => 1 2 4
singlyLinkedList.remove( 1 ); // remove head
singlyLinkedList.print(); // => 2 4
singlyLinkedList.remove( 4 ); // remove tail
singlyLinkedList.print(); // => 2
console.log( 'length is 1:', singlyLinkedList.length() ); // => 1
singlyLinkedList.add( 6 );
singlyLinkedList.print(); // => 2 6
singlyLinkedList.insertAfter( 3, 2 );
singlyLinkedList.print(); // => 2 3 6
singlyLinkedList.insertAfter( 4, 3 );
singlyLinkedList.print(); // => 2 3 4 6
singlyLinkedList.insertAfter( 5, 9 ); // insertAfter a non existing node
singlyLinkedList.print(); // => 2 3 4 6
singlyLinkedList.insertAfter( 5, 4 );
singlyLinkedList.insertAfter( 7, 6 ); // insertAfter the tail
singlyLinkedList.print(); // => 2 3 4 5 6 7
singlyLinkedList.add( 8 ); // add node with normal method
singlyLinkedList.print(); // => 2 3 4 5 6 7 8
console.log( 'length is 7:', singlyLinkedList.length() ); // => 7
singlyLinkedList.traverse( node => {
  node.data = node.data + 10;
} );
singlyLinkedList.print(); // => 12 13 14 15 16 17 18
singlyLinkedList.traverse( node => {
  console.log( node.data );
} ); // => 12 13 14 15 16 17 18
console.log( 'length is 7:', singlyLinkedList.length() ); // => 7

//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
class Graph {
  constructor() {
    this.vertices = [];
    this.edges = [];
    this.numberOfEdges = 0;
  }

  addVertex(vertex) {
    this.vertices.push( vertex );
    this.edges[ vertex ] = [];
  }

  removeVertex(vertex) {
    let index = this.vertices.indexOf( vertex );
    if ( ~index ) {
      this.vertices.splice( index, 1 );
    }
    while ( this.edges[ vertex ].length ) {
      let adjacentVertex = this.edges[ vertex ].pop();
      this.removeEdge( adjacentVertex, vertex );
    }
  }

  addEdge(vertex1, vertex2) {
    this.edges[ vertex1 ].push( vertex2 );
    this.edges[ vertex2 ].push( vertex1 );
    this.numberOfEdges++;
  }

  removeEdge(vertex1, vertex2) {
    let index1 = this.edges[ vertex1 ] ? this.edges[ vertex1 ].indexOf( vertex2 ) : -1;
    let index2 = this.edges[ vertex2 ] ? this.edges[ vertex2 ].indexOf( vertex1 ) : -1;
    if ( ~index1 ) {
      this.edges[ vertex1 ].splice( index1, 1 );
      this.numberOfEdges--;
    }
    if ( ~index2 ) {
      this.edges[ vertex2 ].splice( index2, 1 );
    }
  }
```

```
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
```