```javascript
class MyArray {
  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) {
      return foundIndex;
    }

    return null;
  }

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

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

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

const array = new MyArray();
array.add(1);
array.add(2);
array.add(3);
array.add(4);
array.print(); // => 1 2 3 4
console.log('search 3 gives index 2:', array.search(3)); // => 2
console.log('getAtIndex 2 gives 3:', array.getAtIndex(2)); // => 3
console.log('length is 4:', array.length()); // => 4
array.remove(3);
array.print(); // => 1 2 4
array.add(5);
array.add(5);
array.print(); // => 1 2 4 5 5
array.remove(5);
array.print(); // => 1 2 4

```

```javascript
function Node(data) {
  this.data = data;
  this.left = null;
  this.right = null;
}

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

  add(data) {
    const node = new Node(data);
    if(!this.root) {
      this.root = node;
    } else {
      let current = this.root;
      while(current) {
        if(node.data < current.data) {
          if(!current.left) {
            current.left = node;
            break;
          }
          current = current.left;
        } else if (node.data > current.data) {
          if(!current.right) {
            current.right = node;
            break;
          }
          current = current.right;
        } else {
          break;
        }
      }
    }
  }

  remove(data) {
    const that = this;
    const removeNode = (node, data) => {
      if(!node) {
        return null;
      }
      if(data === node.data) {
        if(!node.left && !node.right) {
          return null;
        }
        if(!node.left) {
          return node.right;
        }
        if(!node.right) {
          return node.left;
        }
        // 2 children
        const temp = that.getMin(node.right);
        node.data = temp;
        node.right = removeNode(node.right, temp);
        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);
  }

  contains(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;
  }

  _preOrder(node, fn) {
    if(node) {
      if(fn) {
        fn(node);
      }
      this._preOrder(node.left, fn);
      this._preOrder(node.right, fn);
    }
  }

  _inOrder(node, fn) {
    if(node) {
      this._inOrder(node.left, fn);
      if(fn) {
        fn(node);
      }
      this._inOrder(node.right, fn);
    }
  }

  _postOrder(node, fn) {
    if(node) {
      this._postOrder(node.left, fn);
      this._postOrder(node.right, fn);
      if(fn) {
        fn(node);
      }
    }
  }

  traverseDFS(fn, method) {
    const current = this.root;
    if(method) {
      this[`_${method}`](current, fn);
    } else {
      this._preOrder(current, fn);
    }
  }

  traverseBFS(fn) {
    this.queue = [];
    this.queue.push(this.root);
    while(this.queue.length) {
      const node = this.queue.shift();
      if(fn) {
        fn(node);
      }
      if(node.left) {
        this.queue.push(node.left);
      }
      if(node.right) {
        this.queue.push(node.right);
      }
    }
  }

  print() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('|');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += `${node.data.toString()} `;
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      if(node.left) {
        queue.push(node.left);
      }
      if(node.right) {
        queue.push(node.right);
      }
    }
    console.log(string.slice(0, -2).trim());
  }

  printByLevel() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('\n');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      if(node.left) {
        queue.push(node.left);
      }
      if(node.right) {
        queue.push(node.right);
      }
    }
    console.log(string.trim());
  }

  getMin(node) {
    if(!node) {
      node = this.root;
    }
    while(node.left) {
      node = node.left;
    }
    return node.data;
  }

  getMax(node) {
    if(!node) {
      node = this.root;
    }
    while(node.right) {
      node = node.right;
    }
    return node.data;
  }

  _getHeight(node) {
    if(!node) {
      return -1;
    }
    const left = this._getHeight(node.left);
    const right = this._getHeight(node.right);
    return Math.max(left, right) + 1;
  }

  getHeight(node) {
    if(!node) {
      node = this.root;
    }
    return this._getHeight(node);
  }

  _isBalanced(node) {
    if(!node) {
      return true;
    }
    const heigthLeft = this._getHeight(node.left);
    const heigthRight = this._getHeight(node.right);
    const diff = Math.abs(heigthLeft - heigthRight);
    if(diff > 1) {
      return false;
    } else {
      return this._isBalanced(node.left) && this._isBalanced(node.right);
    }
  }

  isBalanced(node) {
    if(!node) {
      node = this.root;
    }
    return this._isBalanced(node);
  }

  _checkHeight(node) {
    if(!node) {
      return 0;
    }
    const left = this._checkHeight(node.left);
    if(left === -1) {
      return -1;
    }
    const right = this._checkHeight(node.right);
    if(right === -1) {
      return -1;
    }
    const diff = Math.abs(left - right);
    if(diff > 1) {
      return -1;
    } else {
      return Math.max(left, right) + 1;
    }
  }

  isBalancedOptimized(node) {
    if(!node) {
      node = this.root;
    }
    if(!node) {
      return true;
    }
    if(this._checkHeight(node) === -1) {
      return false;
    } else {
      return true;
    }
  }
}

const binarySearchTree = new BinarySearchTree();
binarySearchTree.add(5);
binarySearchTree.add(3);
binarySearchTree.add(7);
binarySearchTree.add(2);
binarySearchTree.add(4);
binarySearchTree.add(4);
binarySearchTree.add(6);
binarySearchTree.add(8);
binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8
binarySearchTree.printByLevel(); // => 5 \n 3 7 \n 2 4 6 8
console.log('--- DFS inOrder');
binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'inOrder'); // => 2
3 4 5 6 7 8
console.log('--- DFS preOrder');
binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'preOrder'); // =>
5 3 2 4 7 6 8
console.log('--- DFS postOrder');
binarySearchTree.traverseDFS(node => { console.log(node.data); }, 'postOrder'); // =>
2 4 3 6 8 7 5
console.log('--- BFS');
```

```
binarySearchTree.traverseBFS(node => { console.log(node.data); }); // => 5 3 7 2 4 6 8
console.log('min is 2:', binarySearchTree.getMin()); // => 2
console.log('max is 8:', binarySearchTree.getMax()); // => 8
console.log('tree contains 3 is true:', binarySearchTree.contains(3)); // => true
console.log('tree contains 9 is false:', binarySearchTree.contains(9)); // => false
console.log('tree height is 2:', binarySearchTree.getHeight()); // => 2
console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
binarySearchTree.remove(11); // remove non existing node
binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8
binarySearchTree.remove(5); // remove 5, 6 goes up
binarySearchTree.print(); // => 6 | 3 7 | 2 4 8
binarySearchTree.remove(7); // remove 7, 8 goes up
binarySearchTree.print(); // => 6 | 3 8 | 2 4
binarySearchTree.remove(8); // remove 8, the tree becomes unbalanced
binarySearchTree.print(); // => 6 | 3 | 2 4
console.log('tree is balanced is false:', binarySearchTree.isBalanced()); // => true
binarySearchTree.remove(4);
binarySearchTree.remove(2);
binarySearchTree.remove(3);
binarySearchTree.remove(6);
binarySearchTree.print(); // => 'No root node found'
binarySearchTree.printByLevel(); // => 'No root node found'
console.log('tree height is -1:', binarySearchTree.getHeight()); // => -1
console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
console.log('---');
binarySearchTree.add(10);
console.log('tree height is 0:', binarySearchTree.getHeight()); // => 0
console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
binarySearchTree.add(6);
binarySearchTree.add(14);
binarySearchTree.add(4);
binarySearchTree.add(8);
binarySearchTree.add(12);
binarySearchTree.add(16);
binarySearchTree.add(3);
binarySearchTree.add(5);
binarySearchTree.add(7);
binarySearchTree.add(9);
binarySearchTree.add(11);
binarySearchTree.add(13);
binarySearchTree.add(15);
binarySearchTree.add(17);
binarySearchTree.print(); // => 10 | 6 14 | 4 8 12 16 | 3 5 7 9 11 13 15 17
binarySearchTree.remove(10); // remove 10, 11 goes up
binarySearchTree.print(); // => 11 | 6 14 | 4 8 12 16 | 3 5 7 9 x 13 15 17
binarySearchTree.remove(12); // remove 12; 13 goes up
binarySearchTree.print(); // => 11 | 6 14 | 4 8 13 16 | 3 5 7 9 x x 15 17
console.log('tree is balanced is true:', binarySearchTree.isBalanced()); // => true
console.log('tree is balanced optimized is true:',
binarySearchTree.isBalancedOptimized()); // => true
binarySearchTree.remove(13); // remove 13, 13 has no children so nothing changes
binarySearchTree.print(); // => 11 | 6 14 | 4 8 x 16 | 3 5 7 9 x x 15 17
console.log('tree is balanced is false:', binarySearchTree.isBalanced()); // => false
console.log('tree is balanced optimized is false:',
binarySearchTree.isBalancedOptimized()); // => false

```

```javascript
function Node(data) {
  this.data = data;
  this.previous = null;
  this.next = null;
}

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

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

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

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

  traverseReverse(fn) {
    let current = this.tail;
    while(current) {
      if(fn) {
        fn(current);
      }
      current = current.previous;
    }
  }

  length() {
    return this.numberOfValues;
  }

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

const doublyLinkedList = new DoublyLinkedList();
doublyLinkedList.print(); // => ''
doublyLinkedList.add(1);
doublyLinkedList.add(2);
doublyLinkedList.add(3);
doublyLinkedList.add(4);
doublyLinkedList.print(); // => 1 2 3 4
console.log('length is 4:', doublyLinkedList.length()); // => 4
doublyLinkedList.remove(3); // remove value
doublyLinkedList.print(); // => 1 2 4
doublyLinkedList.remove(9); // remove non existing value
doublyLinkedList.print(); // => 1 2 4
doublyLinkedList.remove(1); // remove head
doublyLinkedList.print(); // => 2 4
doublyLinkedList.remove(4); // remove tail
doublyLinkedList.print(); // => 2
console.log('length is 1:', doublyLinkedList.length()); // => 1
```

```
doublyLinkedList.remove(2); // remove tail, the list should be empty
doublyLinkedList.print(); // => ''
console.log('length is 0:', doublyLinkedList.length()); // => 0
doublyLinkedList.add(2);
doublyLinkedList.add(6);
doublyLinkedList.print(); // => 2 6
doublyLinkedList.insertAfter(3, 2);
doublyLinkedList.print(); // => 2 3 6
doublyLinkedList.traverseReverse(node => { console.log(node.data); });
doublyLinkedList.insertAfter(4, 3);
doublyLinkedList.print(); // => 2 3 4 6
doublyLinkedList.insertAfter(5, 9); // insertAfter a non existing node
doublyLinkedList.print(); // => 2 3 4 6
doublyLinkedList.insertAfter(5, 4);
doublyLinkedList.insertAfter(7, 6); // insertAfter the tail
doublyLinkedList.print(); // => 2 3 4 5 6 7
doublyLinkedList.add(8); // add node with normal method
doublyLinkedList.print(); // => 2 3 4 5 6 7 8
console.log('length is 7:', doublyLinkedList.length()); // => 7
doublyLinkedList.traverse(node => { node.data = node.data + 10; });
doublyLinkedList.print(); // => 12 13 14 15 16 17 18
doublyLinkedList.traverse(node => { console.log(node.data); }); // => 12 13 14 15 16 17 18
console.log('length is 7:', doublyLinkedList.length()); // => 7
doublyLinkedList.traverseReverse(node => { console.log(node.data); }); // => 18 17 16 15 14 13 12
doublyLinkedList.print(); // => 12 13 14 15 16 17 18
console.log('length is 7:', doublyLinkedList.length()); // => 7

```

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

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

  removeVertex(vertex) {
    const index = this.vertices.indexOf(vertex);
    if(~index) {
      this.vertices.splice(index, 1);
    }
    while(this.edges[vertex].length) {
      const 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) {
    const index1 = this.edges[vertex1] ? this.edges[vertex1].indexOf(vertex2) : -1;
    const 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);
    }
  }

  size() {
    return this.vertices.length;
  }

  relations() {
    return this.numberOfEdges;
  }

  traverseDFS(vertex, fn) {
    if(!~this.vertices.indexOf(vertex)) {
      return console.log('Vertex not found');
    }
    const visited = [];
    this._traverseDFS(vertex, visited, fn);
  }

  _traverseDFS(vertex, visited, fn) {
    visited[vertex] = true;
    if(this.edges[vertex] !== undefined) {
      fn(vertex);
    }
    for(let i = 0; i < this.edges[vertex].length; i++) {
      if(!visited[this.edges[vertex][i]]) {
        this._traverseDFS(this.edges[vertex][i], visited, fn);
      }
    }
  }

  traverseBFS(vertex, fn) {
    if(!~this.vertices.indexOf(vertex)) {
      return console.log('Vertex not found');
    }
    const queue = [];
    queue.push(vertex);
    const visited = [];
    visited[vertex] = true;

    while(queue.length) {
      vertex = queue.shift();
      fn(vertex);
      for(let i = 0; i < this.edges[vertex].length; i++) {
        if(!visited[this.edges[vertex][i]]) {
          visited[this.edges[vertex][i]] = true;
          queue.push(this.edges[vertex][i]);
        }
      }
    }
  }

  pathFromTo(vertexSource, vertexDestination) {
    if(!~this.vertices.indexOf(vertexSource)) {
      return console.log('Vertex not found');
    }
    const queue = [];
    queue.push(vertexSource);
    const visited = [];
    visited[vertexSource] = true;
    const paths = [];

    while(queue.length) {
      const vertex = queue.shift();
      for(let i = 0; i < this.edges[vertex].length; i++) {
        if(!visited[this.edges[vertex][i]]) {
          visited[this.edges[vertex][i]] = true;
          queue.push(this.edges[vertex][i]);
          // save paths between vertices
          paths[this.edges[vertex][i]] = vertex;
        }
      }
    }
    if(!visited[vertexDestination]) {
      return undefined;
    }

    const path = [];
    for(var j = vertexDestination; j != vertexSource; j = paths[j]) {
      path.push(j);
    }
    path.push(j);
    return path.reverse().join('-');
  }

  print() {
    console.log(this.vertices.map(function(vertex) {
      return (`${vertex} -> ${this.edges[vertex].join(', ')}`).trim();
    }, this).join(' | '));
  }
}

const graph = new Graph();
graph.addVertex(1);
graph.addVertex(2);
graph.addVertex(3);
graph.addVertex(4);
graph.addVertex(5);
graph.addVertex(6);
graph.print(); // 1 -> | 2 -> | 3 -> | 4 -> | 5 -> | 6 ->
graph.addEdge(1, 2);
graph.addEdge(1, 5);
graph.addEdge(2, 3);
graph.addEdge(2, 5);
graph.addEdge(3, 4);
graph.addEdge(4, 5);
graph.addEdge(4, 6);
graph.print(); // 1 -> 2, 5 | 2 -> 1, 3, 5 | 3 -> 2, 4 | 4 -> 3, 5, 6 | 5 -> 1, 2, 4
| 6 -> 4
console.log('graph size (number of vertices):', graph.size()); // => 6
console.log('graph relations (number of edges):', graph.relations()); // => 7
graph.traverseDFS(1, vertex => { console.log(vertex); }); // => 1 2 3 4 5 6
console.log('---');
graph.traverseBFS(1, vertex => { console.log(vertex); }); // => 1 2 5 3 4 6
graph.traverseDFS(0, vertex => { console.log(vertex); }); // => 'Vertex not found'
graph.traverseBFS(0, vertex => { console.log(vertex); }); // => 'Vertex not found'
console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-5-1
console.log('path from 3 to 5:', graph.pathFromTo(3, 5)); // => 3-2-5
graph.removeEdge(1, 2);
graph.removeEdge(4, 5);
graph.removeEdge(10, 11);
console.log('graph relations (number of edges):', graph.relations()); // => 5
console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-3-2-5-1
graph.addEdge(1, 2);
graph.addEdge(4, 5);
console.log('graph relations (number of edges):', graph.relations()); // => 7
console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-5-1
graph.removeVertex(5);
console.log('graph size (number of vertices):', graph.size()); // => 5
console.log('graph relations (number of edges):', graph.relations()); // => 4
console.log('path from 6 to 1:', graph.pathFromTo(6, 1)); // => 6-4-3-2-1
```

```javascript
class HashTable {
  constructor(size) {
    this.values = {};
    this.numberOfValues = 0;
    this.size = size;
  }

  add(key, value) {
    const 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) {
    const 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) {
    const 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(const value in this.values) {
      for(const key in this.values[value]) {
        string += `${this.values[value][key]} `;
      }
    }
    console.log(string.trim());
  }
}

const 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

```

```
class Queue {
  constructor() {
    this.queue = [];
  }

  enqueue(value) {
    this.queue.push(value);
  }

  dequeue() {
    return this.queue.shift();
  }

  peek() {
    return this.queue[0];
  }

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

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

const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
queue.print(); // => 1 2 3
console.log('length is 3:', queue.length()); // => 3
console.log('peek is 1:', queue.peek()); // => 3
console.log('dequeue is 1:', queue.dequeue()); // => 1
queue.print(); // => 2 3
console.log('dequeue is 2:', queue.dequeue());  // => 2
console.log('length is 1:', queue.length()); // => 1
console.log('dequeue is 3:', queue.dequeue()); // => 3
queue.print(); // => ''
console.log('peek is undefined:', queue.peek()); // => undefined
console.log('dequeue is undefined:', queue.dequeue()); // => undefined

```

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

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

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

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

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

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

  difference(set) {
    const 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(' '));
  }
}

const 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('---');
const set1 = new Set();
set1.add(1);
set1.add(2);
const set2 = new Set();
set2.add(2);
set2.add(3);
const set3 = set2.union(set1);
set3.print(); // => 1 2 3
const set4 = set2.intersect(set1);
set4.print(); // => 2
const set5 = set.difference(set3); // 1 2 4 diff 1 2 3
set5.print(); // => 4
const 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) {
    const 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) {
        const 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());
  }
}

const 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

```

```javascript
class Stack {
  constructor() {
    this.stack = [];
  }

  push(value) {
    this.stack.push(value);
  }

  pop() {
    return this.stack.pop();
  }

  peek() {
    return this.stack[this.stack.length - 1];
  }

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

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

const stack = new Stack();
stack.push(1);
stack.push(2);
stack.push(3);
stack.print(); // => 1 2 3
console.log('length is 3:', stack.length()); // => 3
console.log('peek is 3:', stack.peek()); // => 3
console.log('pop is 3:', stack.pop()); // => 3
stack.print(); // => 1 2
console.log('pop is 2:', stack.pop());  // => 2
console.log('length is 1:', stack.length()); // => 1
console.log('pop is 1:', stack.pop()); // => 1
stack.print(); // => ''
console.log('peek is undefined:', stack.peek()); // => undefined
console.log('pop is undefined:', stack.pop()); // => undefined

```

```javascript
function Node(data) {
  this.data = data;
  this.children = [];
}

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

  add(data, toNodeData) {
    const node = new Node(data);
    const parent = toNodeData ? this.findBFS(toNodeData) : null;
    if(parent) {
      parent.children.push(node);
    } else {
      if(!this.root) {
        this.root = node;
      } else {
        return 'Root node is already assigned';
      }
    }
  }

  remove(data) {
    if(this.root.data === data) {
      this.root = null;
    }

    const queue = [this.root];
    while(queue.length) {
      const node = queue.shift();
      for (let [index, child] of node.children.entries()) {
        if(child.data === data) {
          node.children.splice(index, 1);
        } else {
          queue.push(child);
        }
      }
    }
  }

  contains(data) {
    return !!this.findBFS(data);
  }

  findBFS(data) {
    const queue = [this.root];
    while(queue.length) {
      const node = queue.shift();
      if(node.data === data) {
        return node;
      }
      for(const child of node.children) {
        queue.push(child);
      }
    }
    return null;
  }

  _preOrder(node, fn) {
    if(node) {
      if(fn) {
        fn(node);
      }
      for(const child of node.children) {
        this._preOrder(child, fn);
      }
    }
  }

  _postOrder(node, fn) {
    if(node) {
      for(const child of node.children) {
        this._postOrder(child, fn);
      }
      if(fn) {
        fn(node);
      }
    }
  }

  traverseDFS(fn, method) {
    const current = this.root;
    if(method) {
      this[`_${method}`](current, fn);
    } else {
      this._preOrder(current, fn);
    }
  }

  traverseBFS(fn) {
    const queue = [this.root];
    while(queue.length) {
      const node = queue.shift();
      if(fn) {
        fn(node);
      }
      for(const child of node.children) {
        queue.push(child);
      }
    }
  }

  print() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('|');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += `${node.data.toString()} `;
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      for(const child of node.children) {
        queue.push(child);
      }
    }
    console.log(string.slice(0, -2).trim());
  }

  printByLevel() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('\n');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      for(const child of node.children) {
        queue.push(child);
      }
    }
    console.log(string.trim());
  }
}

const tree = new Tree();
tree.add('ceo');
tree.add('cto', 'ceo');
tree.add('dev1', 'cto');
tree.add('dev2', 'cto');
tree.add('dev3', 'cto');
tree.add('cfo', 'ceo');
tree.add('accountant', 'cfo');
tree.add('cmo', 'ceo');
tree.print(); // => ceo | cto cfo cmo | dev1 dev2 dev3 accountant
tree.printByLevel();  // => ceo \n cto cfo cmo \n dev1 dev2 dev3 accountant
console.log('tree contains dev1 is true:', tree.contains('dev1')); // => true
console.log('tree contains dev4 is false:', tree.contains('dev4')); // => false
console.log('--- BFS');
tree.traverseBFS(node => { console.log(node.data); }); // => ceo cto cfo cmo dev1
dev2 dev3 accountant
console.log('--- DFS preOrder');
tree.traverseDFS(node => { console.log(node.data); }, 'preOrder'); // => ceo cto dev1
dev2 dev3 cfo accountant cmo
console.log('--- DFS postOrder');
tree.traverseDFS(node => { console.log(node.data); }, 'postOrder'); // => dev1 dev2
dev3 cto accountant cfo cmo ceo
tree.remove('cmo');
tree.print(); // => ceo | cto cfo | dev1 dev2 dev3 accountant
tree.remove('cfo');
tree.print(); // => ceo | cto | dev1 dev2 dev3

```

```
function Node(data) {
  this.data = data;
  this.isWord = false;
  this.prefixes = 0;
  this.children = {};
}

class Trie {
  constructor() {
    this.root = new Node('');
  }

  add(word) {
    if(!this.root) {
      return null;
    }
    this._addNode(this.root, word);
  }

  _addNode(node, word) {
    if(!node || !word) {
      return null;
    }
    node.prefixes++;
    const letter = word.charAt(0);
    let child = node.children[letter];
    if(!child) {
      child = new Node(letter);
      node.children[letter] = child;
    }
    const remainder = word.substring(1);
    if(!remainder) {
      child.isWord = true;
    }
    this._addNode(child, remainder);
  }

  remove(word) {
    if(!this.root) {
      return;
    }
    if(this.contains(word)) {
      this._removeNode(this.root, word);
    }
  }

  _removeNode(node, word) {
    if(!node || !word) {
      return;
    }
    node.prefixes--;
    const letter = word.charAt(0);

    const child = node.children[letter];
    if(child) {
      const remainder = word.substring(1);
      if(remainder) {
        if(child.prefixes === 1) {
          delete node.children[letter];
        } else {
          this._removeNode(child, remainder);
        }
      } else {
        if(child.prefixes === 0) {
          delete node.children[letter];
        } else {
          child.isWord = false;
        }
      }
    }
  }

  contains(word) {
    if(!this.root) {
      return false;
    }
    return this._contains(this.root, word);
  }

  _contains(node, word) {
    if(!node || !word) {
      return false;
    }
    const letter = word.charAt(0);
    const child = node.children[letter];
    if(child) {
      const remainder = word.substring(1);
      if(!remainder && child.isWord) {
        return true;
      } else {
        return this._contains(child, remainder);
      }
    } else {
      return false;
    }
  }

  countWords() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const queue = [this.root];
    let counter = 0;
    while(queue.length) {
      const node = queue.shift();
      if(node.isWord) {
        counter++;
      }
      for(const child in node.children) {
        if(node.children.hasOwnProperty(child)) {
          queue.push(node.children[child]);
        }
      }
    }
    return counter;
  }

  getWords() {
    const words = [];
    const word = '';
    this._getWords(this.root, words, word);
    return words;
  }

  _getWords(node, words, word) {
    for(const child in node.children) {
      if(node.children.hasOwnProperty(child)) {
        word += child;
        if (node.children[child].isWord) {
          words.push(word);
        }
        this._getWords(node.children[child], words, word);
        word = word.substring(0, word.length - 1);
      }
    }
  }

  print() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('|');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += `${node.data.toString()} `;
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      for(const child in node.children) {
        if(node.children.hasOwnProperty(child)) {
          queue.push(node.children[child]);
        }
      }
    }
    console.log(string.slice(0, -2).trim());
  }

  printByLevel() {
    if(!this.root) {
      return console.log('No root node found');
    }
    const newline = new Node('\n');
    const queue = [this.root, newline];
    let string = '';
    while(queue.length) {
      const node = queue.shift();
      string += node.data.toString() + (node.data !== '\n' ? ' ' : '');
      if(node === newline && queue.length) {
        queue.push(newline);
      }
      for(const child in node.children) {
        if(node.children.hasOwnProperty(child)) {
          queue.push(node.children[child]);
        }
      }
    }
    console.log(string.trim());
  }
```

```
}

const trie = new Trie();
trie.add('one');
trie.add('two');
trie.add('fifth');
trie.add('fifty');
trie.print(); // => | o t f | n w i | e o f | t | h y
trie.printByLevel(); // => o t f \n n w i \n e o f \n t \n h y
console.log('words are: one, two, fifth, fifty:', trie.getWords()); // => [ 'one', 'two', 'fifth', 'fifty' ]
console.log('trie count words is 4:', trie.countWords()); // => 4
console.log('trie contains one is true:', trie.contains('one')); // => true
console.log('trie contains on is false:', trie.contains('on')); // => false
trie.remove('one');
console.log('trie contains one is false:', trie.contains('one')); // => false
console.log('trie count words is 3:', trie.countWords()); // => 3
console.log('words are two, fifth, fifty:', trie.getWords()); // => [ 'two', 'fifth', 'fifty' ]

```