**Practical**

**Data Structures And Algorithms**

**Ashutosh Chandrakant Deshmukh**

**Roll No : 15**

**Division : Technocrats**

1. Print Write a program to test if an array contains a specific value

**Code :**

*var* arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

*var* num = prompt('Enter The Number To Search');

if (arr2.includes(*Number*(num))) {

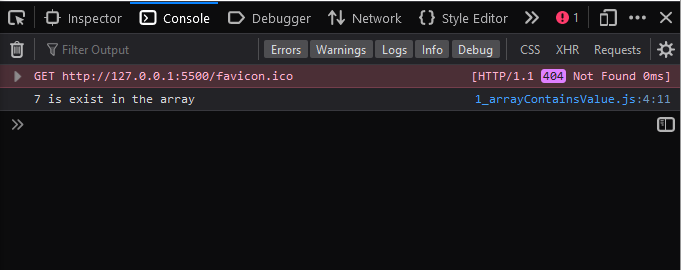
  console.log(num + ' is exist in the array');

} else {

  console.log(num + ' does not exist in the array');

}

**Output :**



1. Write program to accept n numbers from user and print addition of all even numbers.

**Code :**

*function* isEven(*n*) {

  return *n* % 2 == 0;

}

*function* findSum(*no*) {

*let* sum = 0;

  for (*var* i = 1; i <= *no*; i++) {

    if (isEven(i)) {

      sum += i;

    }

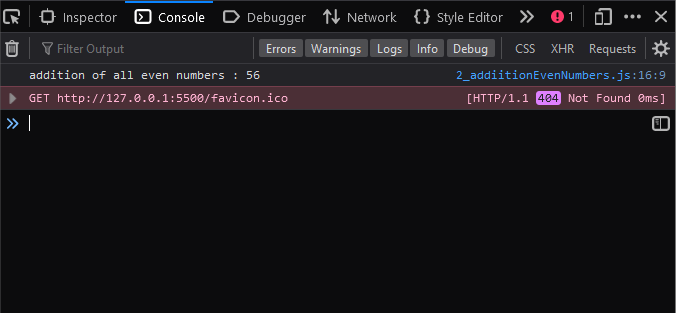
  }

  return sum;

}

console.log('addition of all even numbers : ' + findSum(15));

**Output:**



1. Find the unique element from array

**Code :**

*function* toUniqueArray(*a*) {

*var* newArr = [];

  for (*var* i = 0; i < *a*.length; i++) {

    if (newArr.indexOf(*a*[i]) === -1) {

      newArr.push(*a*[i]);

    }

  }

  return newArr;

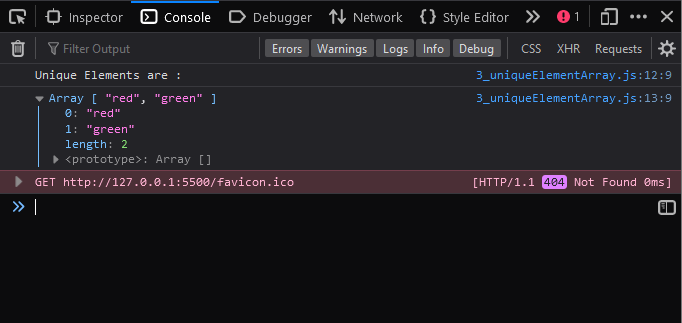
}

*var* array = ['red', 'red', 'green', 'green', 'green'];

*var* uniqueitems = toUniqueArray(array);

console.log(uniqueitems);

**Output :**



1. Write a program which prints frequency of all array elements

**Code :**

*var* arr = [1, 2, 2, 2, 3, 5, 7, 7, 9, 2, 3, 2];

*var* count = 0;

*var* no = prompt('Enter The Element Which Frequency To Check');

for (*let* i = 0; i <= arr.length - 1; i++) {

  if (arr[i] == no) {

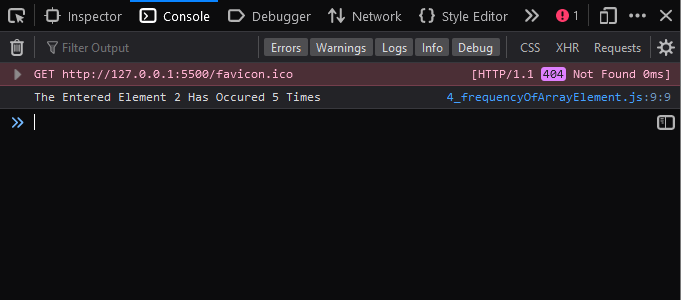
    count++;

  }

}

console.log('The Entered Element ' + no + ' Has Occured ' + count + ' Times');

**Output :**



1. Initialize two arrays with 5 elements, each and find common elements which are common from both.

**Code :**

*function* getCommon(*arr1*, *arr2*) {

*var* common = [];

  for (*var* i = 0; i < *arr1*.length; ++i) {

    for (*var* j = 0; j < *arr2*.length; ++j) {

      if (*arr1*[i] == *arr2*[j]) {

        // If element is in both the arrays

        common.push(*arr1*[i]); // Push to common array

      }

    }

  }

  return common;

}

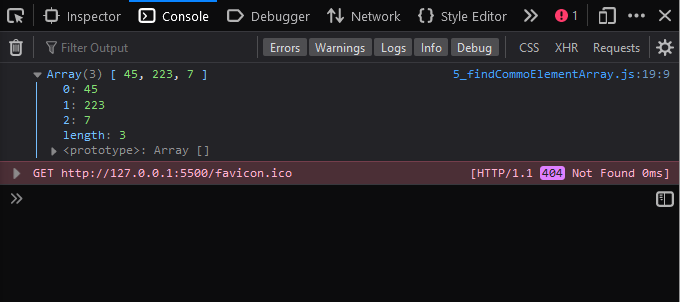
*var* arr1 = [45, 99, 55, 223, 7];

*var* arr2 = [45, 18, 93, 7, 223];

*var* commonElements = getCommon(arr1, arr2);

console.log(commonElements);

**Output :**



1. Write a program which accept matrix from user and find maximum from each column and minimum from each row.

**Code :**

*var* arr = new *Array*();

for (*let* i = 0; i < 2; i++) {

  arr[i] = new *Array*();

  for (*let* j = 0; j < 2; j++) {

    arr[i][j] = *Number*(prompt('Enter Number'));

  }

}

*var* max = arr[0][0];

*var* min = arr[0][0];

for (*var* i = 0; i < 2; i++) {

  for (*var* j = 0; j < 2; j++) {

    if (arr[i][j] > max) {

      max = arr[i][j];

    }

    if (arr[i][j] < min) {

      min = arr[i][j];

    }

  }

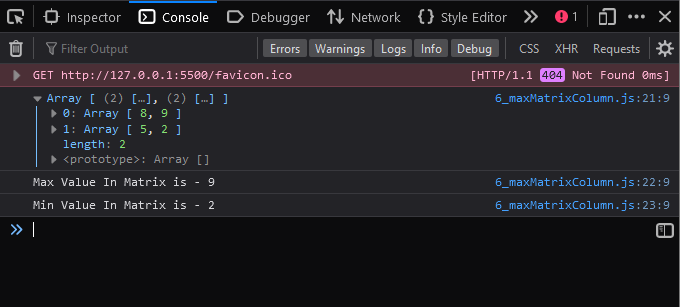
}

console.log(arr);

console.log('Max Value In Matrix is - ' + max);

console.log('Min Value In Matrix is - ' + min);

**Output :**



1. Write a program to implement
   1. Singly linked list with required member function(Create, insert, delete, Display

**Code :**

//creating a node class

*class* Node {

  constructor(*value*) {

    this.data = *value*;

    this.next = null;

  }

}

//implementing linked list using class

*class* SLL {

  constructor() {

    this.head = null; //head is null by default which means list is empty

    this.size = 0; //keep track of size of the list . 0 by default

  }

  //insert first node

  insertFirst(*value*) {

    //create new node

*var* newnode = new *Node*(*value*);

    //if list is empty add the value and make it as first

    if (this.head == null) {

      //document.write("Empty List");

      this.head = newnode;

      this.size++;

    } else {

      newnode.next = this.head;

      this.head = newnode;

      this.size++;

    }

  }

  //insert last node

  insertLast(*value*) {

    //creates new node

*var* newnode = new *Node*(*value*);

*let* current = this.head;

    while (current.next) {

      current = current.next;

    }

    current.next = newnode;

    this.size++;

  }

  //insert at pos

  insertPos(*value*, *pos*) {

*var* newnode = new *Node*(*value*);

*let* current = this.head;

*let* i = 1;

    if (*pos* < 0 || *pos* > this.size) {

      document.write('Invalid Position');

    } else if (*pos* == 1) {

      this.insertFirst(*value*);

    } else if (*pos* == this.count) {

      this.insertLast(*value*);

    } else {

      while (i < *pos* - 1) {

        current = current.next;

        i++;

      }

      newnode.next = current.next;

      current.next = newnode;

      this.size++;

    }

  }

  //delete first

  deletefirst() {

*let* current = this.head;

    if (this.head == null) {

      document.write('Empty');

    } else {

      this.head = current.next;

      current.next = null;

      this.size--;

    }

  }

  //deletelast

  deletelast() {

*let* current = this.head;

    while (current.next.next) {

      current = current.next;

    }

    current.next = null;

    this.count--;

  }

  //delete at position

  deletepos(*pos*) {

*let* i = 1;

*let* current = this.head;

    if (*pos* < 0 || *pos* > this.count) {

      document.write('Invalid position');

    } else if (*pos* == 1) {

      this.deletefirst();

    } else {

      while (i < *pos* - 1) {

        current = current.next;

        i++;

      }

*let* temp = current.next;

      current.next = temp.next;

      temp.next = null;

      this.count--;

    }

  }

  display() {

*let* current = this.head;

    if (current == null) {

      console.log('Empty List');

    } else {

      while (current.next) {

        console.log(current.data + '->');

        current = current.next;

      }

      console.log(current.data + '-> null');

    }

  }

}

*var* SLL1 = new SLL();

SLL1.insertFirst(10);

SLL1.insertFirst(20);

SLL1.insertFirst(30);

SLL1.insertFirst(40);

SLL1.insertLast(50);

SLL1.insertLast(60);

SLL1.insertPos(11, 5);

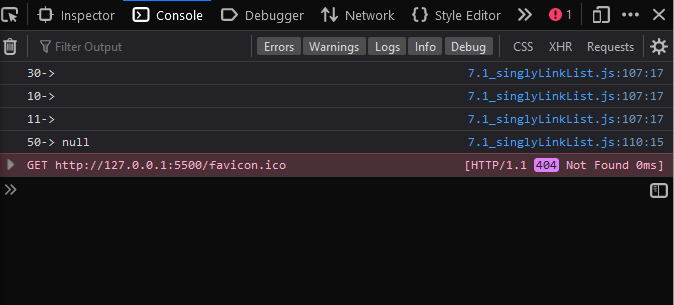
SLL1.deletefirst();

SLL1.deletelast();

SLL1.deletepos(2);

SLL1.display();

**Output :**



1. Write a program to implement
   1. Doubly linked list with required member function(Create, insert, delete, Display

**Code :**

*class* Node {

  constructor(*value*) {

    this.prev = null;

    this.data = *value*;

    this.next = null;

  }

}

*class* doubly {

  constructor() {

    this.head = null;

    this.count = 0;

  }

  //INSERT AT FIRST

  insertatfirst(*val*) {

*var* newnode = new *Node*(*val*);

    if (this.head == null) {

      this.head = newnode;

    } else {

      newnode.next = this.head;

      newnode.prev = null;

      this.head = newnode;

      newnode.next.prev = newnode;

    }

    this.count++;

  }

  //INSERT LAST

  insertatlast(*val*) {

*var* newnode = new *Node*(*val*);

*let* cur = this.head;

    while (cur.next) {

      cur = cur.next;

    }

    newnode.next = null;

    newnode.prev = cur;

    cur.next = newnode;

    this.count++;

  }

  //INSERT BETWEEN

  insertatpos(*val*, *pos*) {

*let* i = 1;

*var* newnode = new *Node*(*val*);

*let* cur = this.head;

    if (*pos* < 0 || *pos* > this.count) {

      alert('invalid position');

    } else if (*pos* == 1) {

      this.insertatfirst(*val*);

    } else if (*pos* == this.count) {

      this.insertatlast(*val*);

    } else {

      while (i < *pos* - 1) {

        cur = cur.next;

        i++;

      }

      newnode.next = cur.next;

      newnode.prev = cur;

      cur.next = newnode;

      newnode.next.prev = newnode;

      this.count++;

    }

  }

  //DELETEFIRST

  deleteatfirst() {

*var* cur = this.head;

    if (this.head == null) {

      alert('emputy');

    } else {

      this.head = cur.next;

      cur.next.prev = null;

      cur.next = null;

      this.count--;

    }

  }

  //DELETELAST

  deleteatlast() {

*var* cur = this.head;

    while (cur.next.next) {

      cur = cur.next;

    }

*var* temp = cur.next;

    cur.next = null;

    temp.prev = null;

    this.count--;

  }

  deleteatpos(*pos*) {

*let* cur = this.head;

*let* i = 1;

    if (*pos* < 0 || *pos* > this.count) {

      alert('invalid position');

    } else if (*pos* == 1) {

      this.deleteatfirst();

    } else if (*pos* == this.count) {

      this.deleteatlast();

    } else {

      while (i < *pos* - 1) {

        cur = cur.next;

        i++;

      }

*var* temp = cur.next;

      cur.next = temp.next;

      temp.next.prev = temp.prev;

      temp = null;

      this.count--;

    }

  }

  display() {

*let* cur = this.head;

    if (this.head == null) {

      console.log('Empty List');

    } else {

      while (cur.next) {

        console.log(cur.data) + '->';

        cur = cur.next;

      }

      console.log(cur.data) + ' -> null';

    }

  }

}

*var* list1 = new doubly();

list1.insertatfirst(10);

list1.insertatfirst(20);

list1.insertatfirst(30);

list1.insertatfirst(40);

list1.insertatlast(50);

list1.insertatlast(60);

// list1.insertatpos(90, 6);

// list1.insertatfirst(56);

//list1.insertatpos(7,7);

//list1.deleteatfirst();

//list1.deleteatlast();

// list1.deleteatlast();

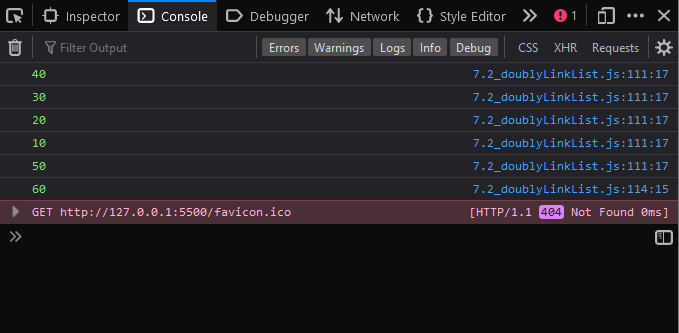
// list1.deleteatpos(3);

// list1.deleteatpos(4);

list1.display();

// console.log(list1.count);

**Output :**



1. Write a program to implement STACK using Array with PUSH, POP operations

**Code :**

*class* Stack {

  constructor(*sz*) {

    this.data = new *Array*();

    this.top = -1;

    this.size = *sz*;

  }

  push(*value*) {

    if (this.top == this.size - 1) {

      console.log('Stack is Full');

    } else {

      this.top++;

      this.data[this.top] = *value*;

    }

  }

  pop() {

    if (this.top == 1) {

      console.log('Stack Is Empty');

    } else {

*let* temp = this.data[this.top];

      this.top--;

      return temp;

    }

  }

  display() {

    for (*let* i = this.top; i >= 0; i--) {

      console.log(this.data[i] + ' ->');

    }

  }

}

*var* stk = new Stack();

stk.push(10);

stk.push(20);

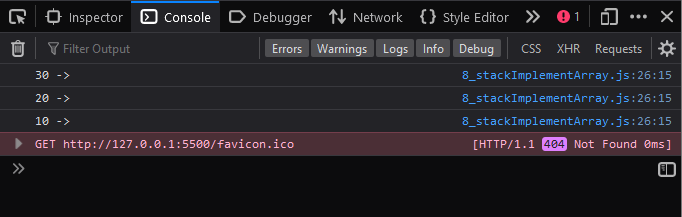
stk.push(30);

stk.push(40);

stk.pop();

stk.display();

**Output :**



1. Write a program to implement Stack using Linked List

**Code :**

*class* Node {

  constructor() {

    this.data = 0;

    this.next = null;

  }

}

*class* Stack {

  constructor() {

    this.top = null;

  }

  push(*x*) {

*let* temp = new *Node*();

    if (temp == null) {

      console.log('Stack is full');

      return;

    }

    temp.data = *x*;

    temp.next = this.top;

    this.top = temp;

  }

  pop() {

    if (this.top == null) {

      console.log('Stack is empty');

      return;

    }

    this.top = this.top.next;

  }

  display() {

    if (this.top == null) {

      console.log('Stack is empty');

    } else {

*let* temp = this.top;

      while (temp != null) {

        console.log(temp.data + '->');

        temp = temp.next;

      }

    }

  }

}

*let* obj = new Stack();

obj.push(11);

obj.push(22);

obj.push(33);

obj.push(44);

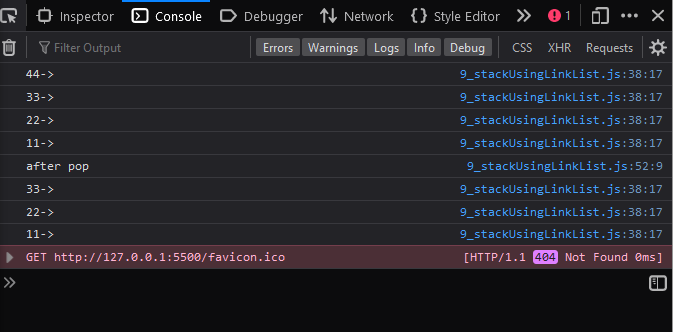
obj.display();

obj.pop();

console.log('after pop');

obj.display();

**Output :**



* 1. Write a application of stack to Check for balanced parentheses.

**Code :**

*class* Stack {

  constructor(*s*) {

    this.data = new *Array*();

    this.top = -1;

    this.size = *s*;

  }

  push(*val*) {

    if (this.top == this.size - 1) {

      document.write('Stack is full');

    } else {

      this.top++;

      this.data.unshift(*val*);

    }

  }

  pop() {

    if (this.top == -1) {

      document.write('Stack is empty');

    } else {

*let* val = this.data.shift();

      this.top--;

      //document.write(val);

    }

  }

  peek() {

    return this.data[this.top];

  }

}

*function* isBalanced(*str*) {

*let* s = new Stack(6);

  for (*let* i = 0; i < *str*.length; i++) {

*let* letter = *str*.charAt(i);

    if (letter === '(' || letter === '[' || letter === '{') {

      s.push(letter);

      //console.log(s);

    } else if (letter === ')' || letter == ']' || letter === '}') {

      if (s.peek() === '(') {

        s.pop();

      } else if (s.peek() === '[') {

        s.pop();

      } else if (s.peek() === '{') {

        s.pop();

      } else return 'Not Balanced expression ';

    }

  }

  if (s.top === -1) {

    return 'Balanced expression ';

  } else {

    return 'Not Balanced expression ';

  }

}

*var* exp1 = '{[a+b](b-a)}';

console.log(exp1);

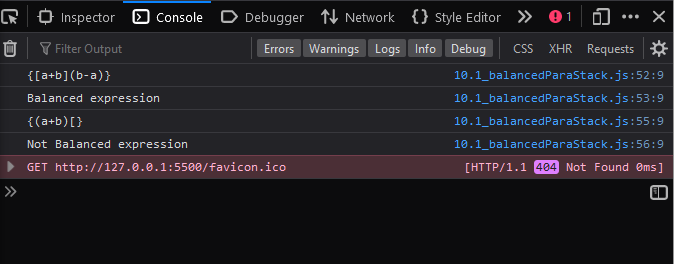
console.log(isBalanced(exp1));

*var* exp2 = '{(a+b)[}';

console.log(exp2);

console.log(isBalanced(exp2));

**Output :**



* 1. Write a program to Reverse a string using stack

**Code :**

*class* Stack {

  constructor() {

    this.elements = [];

  }

  push(*element*) {

    this.elements.push(*element*);

  }

  pop() {

    if (this.elements.length === 0) return 'Underflow situation';

    else return this.elements.pop();

  }

  isEmpty() {

    if (this.elements.length > 0) return false;

    else return true;

  }

}

*function* reverse(*str*) {

  //Creates a new stack

*let* stack = new Stack();

*let* i = 0;

*let* reversedStr = '';

  //Adds all the characters to the stack.

  while (i !== *str*.length) {

    stack.push(*str*.charAt(i));

    i++;

  }

  //Creates a reversed string by popping the stack.

  while (!stack.isEmpty()) {

    reversedStr += stack.pop();

  }

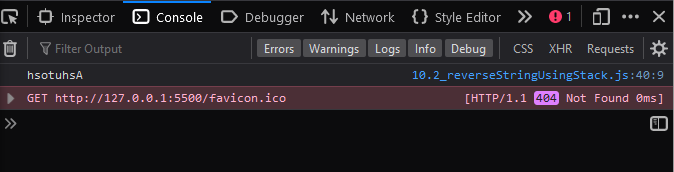
  //returns the reversed string.

  return reversedStr;

}

console.log(reverse('Ashutosh'));

**Output :**



1. Write a program to implement Linear Queue.

**Code :**

// Queue class

*class* Queue {

  // Array is used to implement a Queue

  constructor() {

    this.items = [];

  }

  enqueue(*element*) {

    // adding element to the queue

    this.items.push(*element*);

  }

  dequeue() {

    if (this.isEmpty()) return 'Underflow';

    return this.items.shift();

  }

  front() {

    if (this.isEmpty()) return 'No elements in Queue';

    return this.items[0];

  }

  isEmpty() {

    // return true if the queue is empty.

    return this.items.length == 0;

  }

  // printQueue function

  printQueue() {

*var* str = '';

    for (*var* i = 0; i < this.items.length; i++) str += this.items[i] + ' ';

    return str;

  }

}

// creating object for queue class

*var* queue = new Queue();

// Testing dequeue and pop on an empty queue returns Underflow

console.log(queue.dequeue());

// returns true

console.log(queue.isEmpty());

// Adding elements to the queue contains [10, 20, 30, 40, 50]

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

queue.enqueue(50);

queue.enqueue(60);

console.log(queue.front());

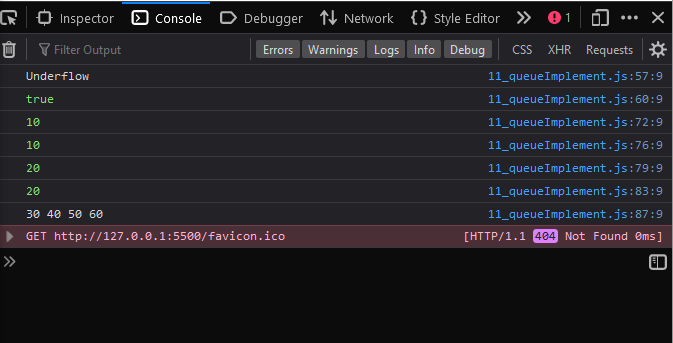
console.log(queue.dequeue());

console.log(queue.front());

console.log(queue.dequeue());

console.log(queue.printQueue());

**Output :**



1. Write a program to implement Circular Queue

**Code :**

*class* CQ {

  constructor(sz) {

    this.data = new Array();

    this.front = -1;

    this.rear = -1;

    this.size = sz;

  }

  enqueue(value) {

    if (

      (this.front == 0 && this.rear == this.size - 1) ||

      this.rear == this.front - 1

    ) {

*console*.log('Queue is Full');

    } else {

      if (this.rear == -1) {

        this.rear++;

        this.front++;

        this.data[this.rear] = value;

      } else if (this.rear == this.size - 1) {

        this.rear = 0;

        this.data[this.rear] = value;

      } else {

        this.rear++;

        this.data[this.rear] = value;

      }

    }

  }

  dequeue() {

    if (this.rear == -1) {

*console*.log('Queue is Empty');

    } else {

      if (this.front == this.size - 1) {

        value = this.data[this.front];

        this.front = 0;

        return value;

      } else if (this.front == this.rear) {

        value = this.data[this.front];

        this.front = -1;

        this.rear = -1;

        return value;

      } else {

*var* value = this.data[this.front];

        this.front++;

        return value;

      }

    }

  }

  display() {

    if (this.rear == -1) {

      console.log('Queue is Empty');

    } else {

      if (this.rear >= this.front) {

        console.log('Elements In The Circular Queue: ');

        for (*var* i = this.front; i <= this.rear; i++) {

          console.log(this.data[i] + '->');

        }

      }

      // If rear crossed the max index and indexing has started in loop

      else {

        console.log('Elements In The Circular  Queue:');

        // Loop for printing elements from front to max size or last index

        for (*var* i = this.front; i <= this.size; i++) {

          console.log(this.data[i] + ' -->');

        }

        // Loop for printing elements from 0th index till rear position

        for (*var* i = 0; i <= this.rear; i++) {

          console.log(this.data[i] + ' --->');

        }

      }

    }

  }

}

*var* cqueue = new CQ(5);

cqueue.enqueue(10);

cqueue.enqueue(20);

cqueue.enqueue(30);

cqueue.enqueue(40);

cqueue.enqueue(50);

cqueue.display();

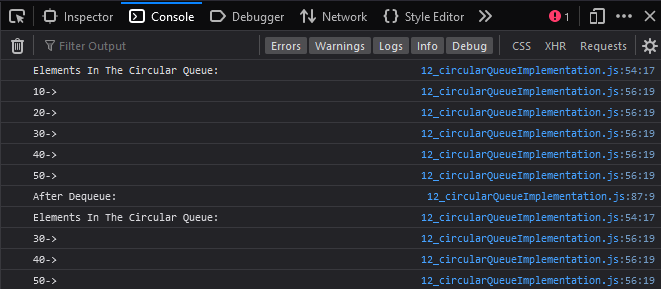
cqueue.dequeue();

cqueue.dequeue();

console.log('After Dequeue:');

cqueue.display();

**Output :**



1. Write a program to implement Priority Queue

**Code :**

*class* QElement {

  constructor(element, priority) {

    this.element = element;

    this.priority = priority;

  }

}

*class* PriorityQueue {

  constructor() {

    this.items = [];

  }

  enqueue(element, priority) {

    // creating object from queue element

*var* qElement = new QElement(element, priority);

*var* contain = false;

    for (*var* i = 0; i < this.items.length; i++) {

      if (this.items[i].priority > qElement.priority) {

        // Once the correct location is found it is

        // enqueued

        this.items.splice(i, 0, qElement);

        contain = true;

        break;

      }

    }

    if (!contain) {

      this.items.push(qElement);

    }

  }

  dequeue() {

    if (this.isEmpty()) return 'Underflow';

    return this.items.shift();

  }

  front() {

    // returns the highest priority element in the Priority queue without removing it.

    if (this.isEmpty()) return 'No elements in Queue';

    return this.items[0];

  }

  rear() {

    // returns the lowest priority lement of the queue

    if (this.isEmpty()) return 'No elements in Queue';

    return this.items[this.items.length - 1];

  }

  isEmpty() {

    return this.items.length == 0;

  }

  printPQueue() {

*var* str = '';

    for (*var* i = 0; i < this.items.length; i++)

      str += this.items[i].element + ' ';

    return str;

  }

}

*var* priorityQueue = new PriorityQueue();

console.log(priorityQueue.isEmpty());

console.log(priorityQueue.front());

priorityQueue.enqueue('Ashutosh', 2);

priorityQueue.enqueue('Swanand', 1);

priorityQueue.enqueue('Saurabh', 1);

priorityQueue.enqueue('Shreyash', 2);

priorityQueue.enqueue('Gaurav', 3);

console.log(priorityQueue.printPQueue());

console.log(priorityQueue.front().element);

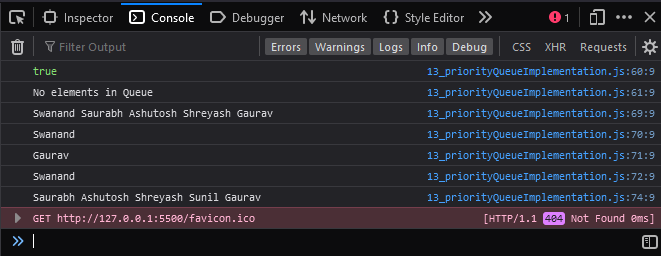
console.log(priorityQueue.rear().element);

console.log(priorityQueue.dequeue().element);

priorityQueue.enqueue('Sunil', 2);

console.log(priorityQueue.printPQueue());

**Output :**



1. Write a program to Reverse stack using queue

**Code :**

*class* Stack {

  constructor() {

    this.elements = [];

  }

  push(element) {

    this.elements.push(element);

  }

  pop() {

    if (this.isEmpty()) return 'Underflow situation';

    else return this.elements.pop();

  }

  isEmpty() {

    return this.elements.length == 0;

  }

  print() {

    return this.elements;

  }

}

*class* Queue {

  constructor() {

    this.elements = [];

  }

  enqueue(*element*) {

    this.elements.push(*element*);

  }

  dequeue() {

    if (!this.isEmpty()) {

      return this.elements.shift();

    } else {

      return 'Underflow situation';

    }

  }

  isEmpty() {

    return this.elements.length == 0;

  }

}

*function* reverse(*stack*) {

*const* queue = new Queue(); //Creating a new queue

  while (!*stack*.isEmpty()) {

    // Looping until the stack is empty.

    queue.enqueue(*stack*.pop()); //Enqueue the popped stack element.

  }

  while (!queue.isEmpty()) {

    //Looping until the queue is empty.

*stack*.push(queue.dequeue()); //Pushed the dequeued queue element.

  }

}

//Creates a new stack.

*const* stack = new Stack();

//Pushes 3 strings onto the stack

stack.push('Welcome');

stack.push('There');

stack.push('Hi');

//Prints the stack before the reversal.

console.log('Printing stack before reversal: ', stack.print());

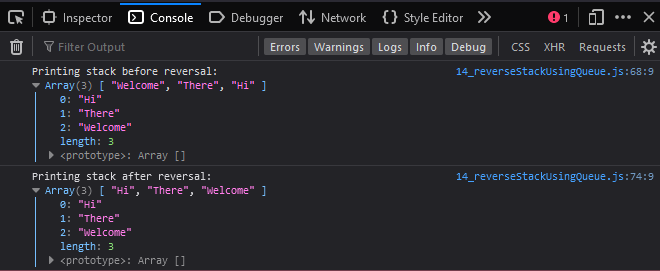
//Reverses the stack.

reverse(stack);

//Prints the stack after the reversal.

console.log('Printing stack after reversal: ', stack.print());

**Output :**



1. Write a program to implement binary search tree with its Traversal

**Code :**

*class* Node {

  constructor(value) {

    this.data = value;

    // this.value = value;

    this.left = null;

    this.right = null;

  }

}

*class* BinarySeachTree {

  constructor() {

    this.root = null;

  }

  insert(value) {

*var* nnode = new Node(value);

    if (this.root == null) {

      this.root = nnode;

      return;

    } else {

*var* tempRoot = this.root;

      while (true) {

        if (tempRoot.data > nnode.data) {

          if (tempRoot.left == null) {

            tempRoot.left = nnode;

            return;

          }

          tempRoot = tempRoot.left;

        } else {

          if (tempRoot.right == null) {

            tempRoot.right = nnode;

            return;

          } else {

            tempRoot = tempRoot.right;

          }

        }

      }

    }

  }

  find(value) {

    if (!this.root) return false;

*var* current = this.root;

*var* found = false;

    while (current && !found) {

      if (*value* < current.value) {

        current = current.left;

      } else if (*value* > current.value) {

        current = current.right;

      } else {

        found = current;

      }

    }

    if (!found) return undefined;

    return found;

  }

  remove(*value*) {

    this.root = this.deleteNode(this.root, *value*);

  }

  deleteNode(*current*, *value*) {

    if (*current* === null) return *current*;

    if (*value* === *current*.value) {

      if (*current*.left === null && *current*.right === null) {

        return null;

      } else if (*current*.left === null) {

        return *current*.right;

      } else if (*current*.right === null) {

        return *current*.left;

      } else {

*let* tempNode = this.kthSmallestNode(*current*.right);

*current*.value = tempNode.value;

*current*.right = this.deleteNode(*current*.right, tempNode.value);

        return *current*;

      }

    } else if (*value* < *current*.value) {

*current*.left = this.deleteNode(*current*.left, *value*);

      return *current*;

    } else {

*current*.right = this.deleteNode(*current*.right, *value*);

      return *current*;

    }

  }

  search(*data*, *key*) {

    if (this.data === null) {

      console.log('Tree is Empty');

    }

    else if (*key* < this.data.key) {

      return this.search(this.data.left, this.data);

    }

    else if (*key* > this.data.key) {

      return this.search(this.data.right, *key*);

    }

    else {

      console.log('Value is equal to root ');

      return *data*;

    }

  }

display() {

    this.inorder(this.root);

  }

  inorder(*root*) {

*let* current = *root*;

    if (current != null) {

      this.inorder(current.left);

      console.log(current.data + ' ->');

      this.inorder(current.right);

    }

  }

}

*var* bst = new BinarySeachTree();

bst.insert(22);

bst.insert(11);

bst.insert(12);

bst.insert(28);

bst.insert(21);

bst.insert(23);

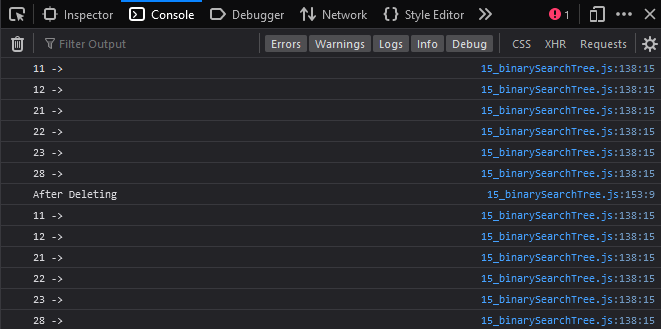
bst.display();

console.log('After Deleting ');

bst.remove(21);

bst.display();

**Output :**



1. Write a program to implement AVL Tree.

**Code :**

// Create node

*const* Node = *function* (*item*) {

  this.item = *item*;

  this.height = 1;

  this.left = null;

  this.right = null;

};

//AVL Tree

*const* AVLTree = *function* () {

*let* root = null;

  //return height of the node

  this.height = (*N*) *=>* {

    if (*N* === null) {

      return 0;

    }

    return *N*.height;

  };

  //right rotate

  this.rightRotate = (*y*) *=>* {

*let* x = *y*.left;

*let* T2 = x.right;

    x.right = *y*;

*y*.left = T2;

*y*.height = Math.max(this.height(*y*.left), this.height(*y*.right)) + 1;

    x.height = Math.max(this.height(x.left), this.height(x.right)) + 1;

    return x;

  };

  //left rotate

  this.leftRotate = (*x*) *=>* {

*let* y = *x*.right;

*let* T2 = y.left;

    y.left = *x*;

*x*.right = T2;

*x*.height = Math.max(this.height(*x*.left), this.height(*x*.right)) + 1;

    y.height = Math.max(this.height(y.left), this.height(y.right)) + 1;

    return y;

  };

  this.getBalanceFactor = (*N*) *=>* {

    if (*N* == null) {

      return 0;

    }

    return this.height(*N*.left) - this.height(*N*.right);

  };

*const* insertNodeHelper = (*node*, *item*) *=>* {

    if (node === null) {

      return new Node(item);

    }

    if (item < node.item) {

      node.left = insertNodeHelper(node.left, item);

    } else if (item > node.item) {

      node.right = insertNodeHelper(node.right, item);

    } else {

      return node;

    }

    node.height = 1 + *Math*.max(this.height(node.left), this.height(node.right));

*let* balanceFactor = this.getBalanceFactor(node);

    if (balanceFactor > 1) {

      if (item < node.left.item) {

        return this.rightRotate(node);

      } else if (item > node.left.item) {

        node.left = this.leftRotate(node.left);

        return this.rightRotate(node);

      }

    }

    if (balanceFactor < -1) {

      if (item > node.right.item) {

        return this.leftRotate(node);

      } else if (item < node.right.item) {

        node.right = this.rightRotate(node.right);

        return this.leftRotate(node);

      }

    }

    return node;

  };

  this.insertNode = (item) *=>* {

    // console.log(root);

*root* = insertNodeHelper(*root*, item);

  };

  this.nodeWithMimumValue = (node) *=>* {

*let* current = node;

    while (current.left !== null) {

      current = current.left;

    }

    return current;

  };

*const* deleteNodeHelper = (*root*, item) *=>* {

    if (*root* == null) {

      return *root*;

    }

    if (item < *root*.item) {

*root*.left = deleteNodeHelper(*root*.left, item);

    } else if (item > *root*.item) {

*root*.right = deleteNodeHelper(*root*.right, item);

    } else {

      if (*root*.left === null || *root*.right === null) {

*let* temp = null;

        if (temp == *root*.left) {

          temp = *root*.right;

        } else {

          temp = *root*.left;

        }

        if (temp == null) {

          temp = *root*;

*root* = null;

        } else {

*root* = temp;

        }

      } else {

*let* temp = this.nodeWithMimumValue(*root*.right);

*root*.item = temp.item;

*root*.right = deleteNodeHelper(*root*.right, temp.item);

      }

    }

    if (*root* == null) {

      return *root*;

    }

*root*.height = *Math*.max(this.height(*root*.left), this.height(*root*.right)) + 1;

*let* balanceFactor = this.getBalanceFactor(*root*);

    if (balanceFactor > 1) {

      if (this.getBalanceFactor(*root*.left) >= 0) {

        return this.rightRotate(*root*);

      } else {

*root*.left = this.leftRotate(*root*.left);

        return this.rightRotate(*root*);

      }

    }

    if (balanceFactor < -1) {

      if (this.getBalanceFactor(*root*.right) <= 0) {

        return this.leftRotate(*root*);

      } else {

*root*.right = this.rightRotate(*root*.right);

        return this.leftRotate(*root*);

      }

    }

    return *root*;

  };

  this.deleteNode = (item) *=>* {

*root* = deleteNodeHelper(*root*, item);

  };

  this.preOrder = () *=>* {

    preOrderHelper(root);

  };

*const* preOrderHelper = (*node*) *=>* {

    if (*node*) {

      console.log(*node*.item);

      preOrderHelper(*node*.left);

      preOrderHelper(*node*.right);

    }

  };

};

*var* tree = new AVLTree();

tree.insertNode(33);

tree.insertNode(13);

tree.insertNode(53);

tree.insertNode(9);

tree.insertNode(21);

tree.insertNode(61);

tree.insertNode(8);

tree.insertNode(11);

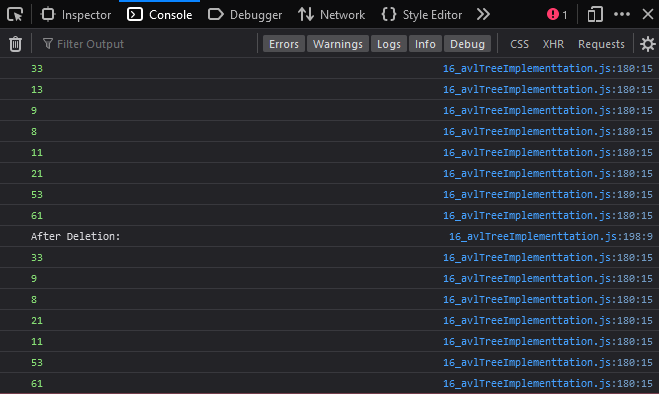
tree.preOrder();

tree.deleteNode(13);

console.log('After Deletion: ');

tree.preOrder();

**Output :**



1. Write a program to implement Graph and display it using different types of graph traversals(BFS,DFS)

**Code :**

*class* Graph {

  constructor() {

    this.nodes = new *Map*();

  }

  addNode(*node*) {

    this.nodes.set(*node*, []);

  }

  addEdge(*source*, *destination*) {

    this.nodes.get(*source*).push(*destination*);

    this.nodes.get(*destination*).push(*source*);

  }

  removeNode(*node*) {

*let* neighbors = this.nodes.get(*node*);

    for (*let* neighbor of neighbors) {

*let* adjacencyListOfNeighbor = this.nodes.get(neighbor);

      this.getIndexAndRemoveItem(*node*, adjacencyListOfNeighbor);

    }

    this.nodes.delete(*node*);

  }

  removeEdge(*source*, *destination*) {

*let* adjacencyListOfSource = this.nodes.get(*source*);

*let* adjacencyListOfDestination = this.nodes.get(*destination*);

    this.getIndexAndRemoveItem(*source*, adjacencyListOfDestination);

    this.getIndexAndRemoveItem(*destination*, adjacencyListOfSource);

  }

  getIndexAndRemoveItem(*item*, *list*) {

*const* index = *list*.indexOf(*item*);

*list*.splice(index, 1);

  }

  depthFirstSearch(*startingNode*) {

*let* visitedNode = [];

    this.dfsRecursion(*startingNode*, visitedNode);

  }

  dfsRecursion(*currentNode*, *visitedNode*) {

*visitedNode*[*currentNode*] = true;

    console.log(*currentNode*);

*let* adjacencyListOfCurrentNode = this.nodes.get(*currentNode*);

    for (*var* node of adjacencyListOfCurrentNode) {

      if (!visitedNode[node]) this.dfsRecursion(node, visitedNode);

    }

  }

  breadthFirstSearch(startingNode) {

*let* visitedNode = [];

*let* queue = [];

    visitedNode[startingNode] = true;

    queue.push(startingNode);

    while (queue.length > 0) {

*const* currentNode = queue.shift();

*console*.log(currentNode);

*const* adjacencyListOfCurrentNode = this.nodes.get(currentNode);

      for (*let* node of adjacencyListOfCurrentNode) {

        if (!visitedNode[node]) {

          visitedNode[node] = true;

          queue.push(node);

        }

      }

    }

  }

  display() {

    for (*let* [node, adjacencyList] of this.nodes) {

*console*.log(`${node}: ${adjacencyList}`);

    }

  }

}

*let* graph = new Graph();

// intially nodes is empty

// add nodes

graph.addNode('A');

graph.addNode('B');

graph.addNode('D');

graph.addNode('C');

// graph.display()

// add edges

graph.addEdge('A', 'B');

graph.addEdge('A', 'D');

graph.addEdge('B', 'C');

graph.addEdge('B', 'D');

console.log('Graph : ');

graph.display();

// depth-first-search

*console*.log('depth-first-search');

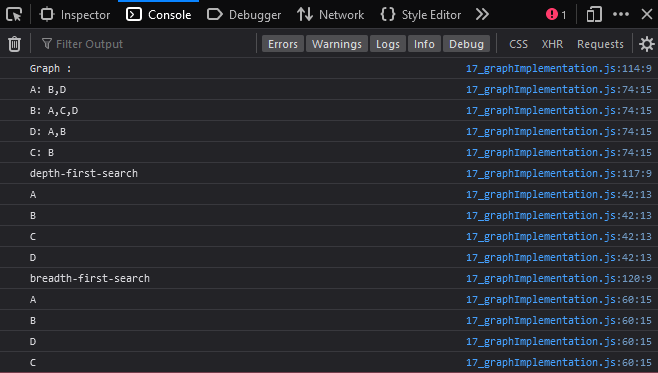
graph.depthFirstSearch('A');

// breadth-first-search

*console*.log('breadth-first-search');

graph.breadthFirstSearch('A');

**Output :**



1. Write a program to implement Brute Force technique
2. **Linear Search**

**Code** :

// Brutefroce - Linear Search

*class* search {

  linear\_search(*arr*, *key*) {

*var* len = *arr*.length;

*var* flag = 0;

    for (*var* i = 0; i < len; i++) {

      if (*arr*[i] == *key*) {

        console.log('Element found!');

        flag = 1;

        break;

      }

    }

    if (flag == 0) console.log('Element not found!');

  }

}

// Execute Linear search

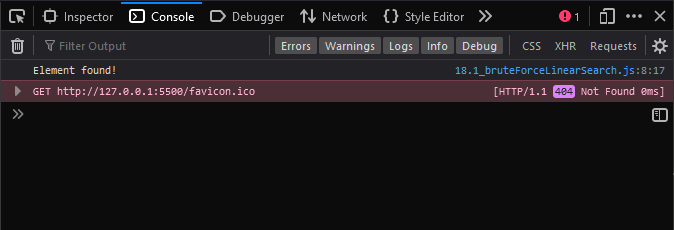
*var* s = new search();

*var* arr = [1, 3, 5, 7, 8, 9];

*var* key = 7;

s.linear\_search(arr, key);

**Output** :



1. **Max Sub Array**

**Code** :

*var* maxSubArray = *function* (*nums*) {

*let* currSum = *nums*[0];

*let* maxSum = currSum;

  for (*let* i = 0; i < *nums*.length; i++) {

    currSum = *nums*[i];

    if (currSum > maxSum) {

      maxSum = currSum;

    }

    for (*let* j = i + 1; j < *nums*.length; j++) {

      currSum = currSum + *nums*[j];

      if (currSum > maxSum) {

        maxSum = currSum;

      }

    }

  }

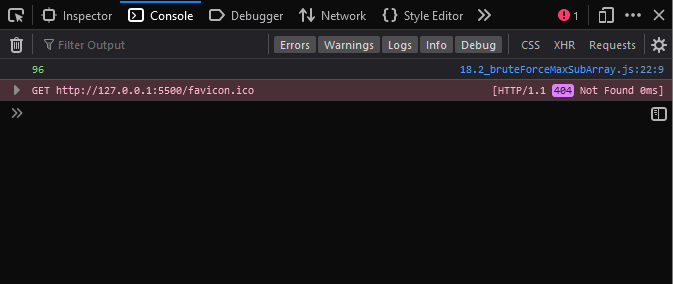
  return maxSum;

};

*var* arr = [2, 4, 5, 7, 12, 3, 1, 11, 13, 11, 15, 12];

console.log(maxSubArray(arr));

**Output** :



1. **Rain Terrace**

**Code** :

*function* maxWater(*arr*, *n*) {

  // To store the maximum water that can be stored

*let* res = 0;

  // For every element of the array except first and last element

  for (*let* i = 1; i < *n* - 1; i++) {

    // Find maximum element on its left

*let* left = *arr*[i];

    for (*let* j = 0; j < i; j++) {

      left = Math.max(left, *arr*[j]);

    }

    // Find maximum element on its right

*let* right = *arr*[i];

    for (*let* j = i + 1; j < *n*; j++) {

      right = Math.max(right, *arr*[j]);

    }

    // Update maximum water value

    res += Math.min(left, right) - *arr*[i];

  }

  return res;

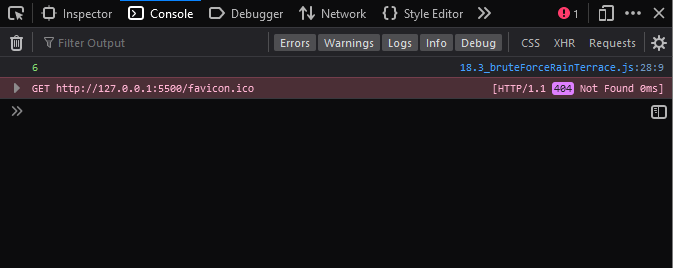
}

*let* arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1];

*let* n = arr.length;

console.log(maxWater(arr, n));

**Output :**



1. **Rain Terrace**

**Code** :

*const* recursiveStaircase = (*num* = 10) *=>* {

  if (*num* <= 0) {

    return 0;

  }

*const* steps = [1, 2];

  if (*num* <= 2) {

    return steps[*num* - 1];

  }

  for (*let* currentStep = 3; currentStep <= *num*; currentStep += 1) {

    [steps[0], steps[1]] = [steps[1], steps[0] + steps[1]];

  }

  return steps[1];

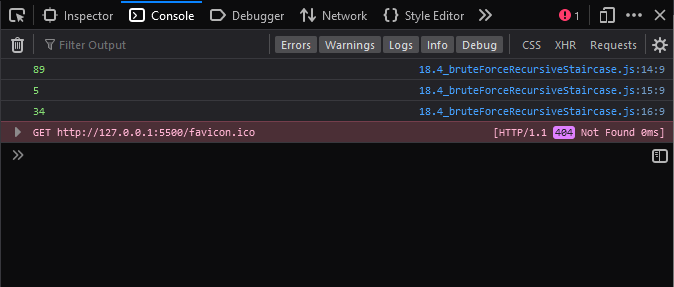
};

console.log(recursiveStaircase());

console.log(recursiveStaircase(4));

console.log(recursiveStaircase(8));

**Output** :



1. Write a program to implement Greedy Algorithm-Prim’s

**Code :**

// Number of vertices in the graph

*let* V = 5;

// A utility function to find the vertex with minimum key value, from the set of vertices

not yet included in MST

*function* minKey(key, mstSet) {

  // Initialize min value

*let* min = *Number*.MAX\_VALUE,

    min\_index;

  for (*let* v = 0; v < V; v++)

    if (mstSet[v] == false && key[v] < min) (min = key[v]), (min\_index = v);

  return min\_index;

}

// A utility function to print the constructed MST stored in parent[]

*function* printMST(parent, graph) {

*console*.log('Edge   Weight');

  for (*let* i = 1; i < V; i++)

    console.log(*parent*[i] + '  - ' + i + '  ' + *graph*[i][*parent*[i]]);

}

// Function to construct and print MST for a graph represented using adjacency matrix representation

*function* primMST(*graph*) {

  // Array to store constructed MST

*let* parent = [];

  // Key values used to pick minimum weight edge in cut

*let* key = [];

  // To represent set of vertices included in MST

*let* mstSet = [];

  // Initialize all keys as INFINITE

  for (*let* i = 0; i < V; i++) (key[i] = *Number*.MAX\_VALUE), (mstSet[i] = false);

  // Always include first 1st vertex in MST. Make key 0 so that this vertex is picked as first vertex.

  key[0] = 0;

  parent[0] = -1; // First node is always root of MST

  // The MST will have V vertices

  for (*let* count = 0; count < V - 1; count++) {

    // Pick the minimum key vertex from the set of vertices not yet included in MST

*let* u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set

    mstSet[u] = true;

    for (*let* v = 0; v < V; v++)

      if (*graph*[u][v] && mstSet[v] == false && *graph*[u][v] < key[v])

        (parent[v] = u), (key[v] = *graph*[u][v]);

  }

  printMST(parent, *graph*);

}

*let* graph = [

  [0, 2, 0, 6, 0],

  [2, 0, 3, 8, 5],

  [0, 3, 0, 0, 7],

  [6, 8, 0, 0, 9],

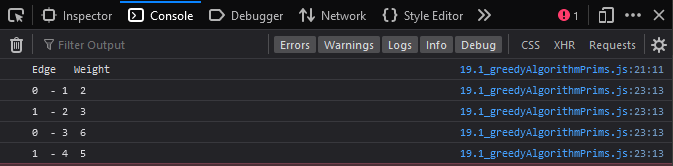
  [0, 5, 7, 9, 0],

];

// Print the solution

primMST(graph);

**Output :**



1. Write a program to implement Divide and Conquer Technique –

**A. Tower of Hanoi**

**Code :**

*function* towerOfHanoi(*n*, *from\_rod*, *to\_rod*, *aux\_rod*) {

  if (*n* == 0) {

    return;

  }

  towerOfHanoi(*n* - 1, *from\_rod*, *aux\_rod*, *to\_rod*);

  console.log(

    'Move disk ' + *n* + ' from rod ' + *from\_rod* + ' to rod ' + *to\_rod* + '<br/>'

  );

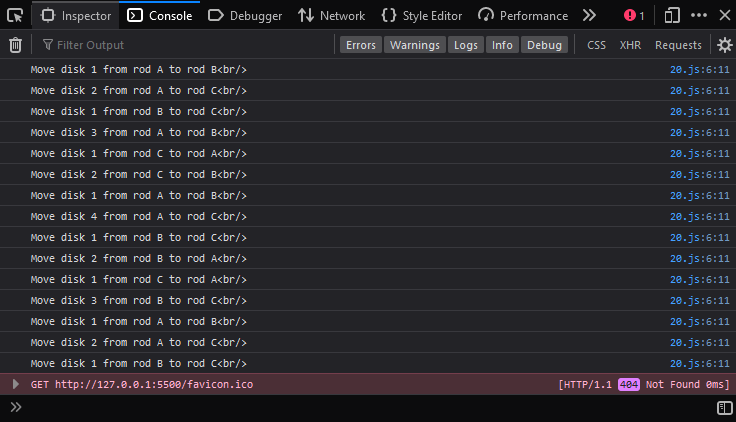
  towerOfHanoi(*n* - 1, *aux\_rod*, *to\_rod*, *from\_rod*);

}

// Driver code

*var* n = 4; // Number of disks

towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods

**Output :**

**B. Binary Search**

**Code :**

*let* recursiveFunction = *function* (*arr*, *x*, *start*, *end*) {

  if (*start* > *end*) return false;

*let* mid = Math.floor((*start* + *end*) / 2);

  if (*arr*[mid] === *x*) return true;

  if (*arr*[mid] > *x*) return recursiveFunction(*arr*, *x*, *start*, mid - 1);

  else return recursiveFunction(*arr*, *x*, mid + 1, *end*);

};

*let* arr = [1, 3, 5, 7, 8, 9];

*let* x = 5;

if (recursiveFunction(arr, x, 0, arr.length - 1))

  console.log('Element found!<br>');

else console.log('Element not found!<br>');

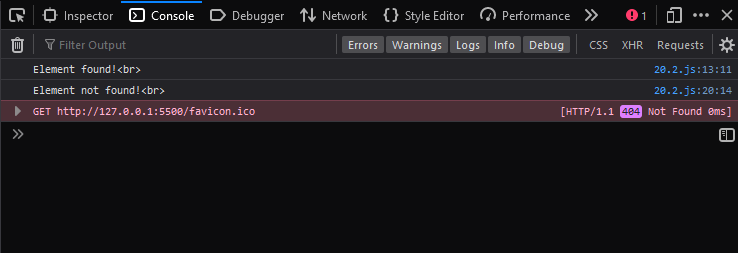
x = 6;

if (recursiveFunction(arr, x, 0, arr.length - 1))

  console.log('Element found!<br>');

else console.log('Element not found!<br>');

**Output :**



1. Write a program to implement Dynamic Programming- LCS

**Code :**

*function* lcs(*X*, *Y*, *m*, *n*) {

  if (*m* == 0 || *n* == 0) return 0;

  if (*X*[*m* - 1] == *Y*[*n* - 1]) return 1 + lcs(*X*, *Y*, *m* - 1, *n* - 1);

  else return max(lcs(*X*, *Y*, *m*, *n* - 1), lcs(*X*, *Y*, *m* - 1, *n*));

}

*function* max(*a*, *b*) {

  return *a* > *b* ? *a* : *b*;

}

*var* s1 = 'AGGTAB';

*var* s2 = 'GXTXAYB';

*var* X = s1;

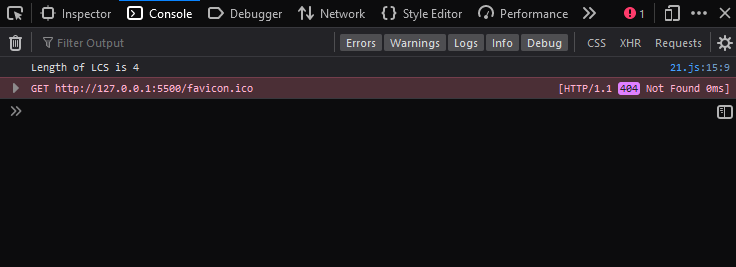
*var* Y = s2;

*var* m = X.length;

*var* n = Y.length;

console.log('Length of LCS is' + ' ' + lcs(X, Y, m, n));

**Output :**



1. Write a program to implement Dynamic Programming- Regular Expression Matching

**Code :**

*function* myFunction() {

  // input string

*var* str = 'Visit me in college!';

  // searching string with modifier i

*var* n = str.search(/me in college/i);

  console.log(n + ' ');

  // searching string without modifier i

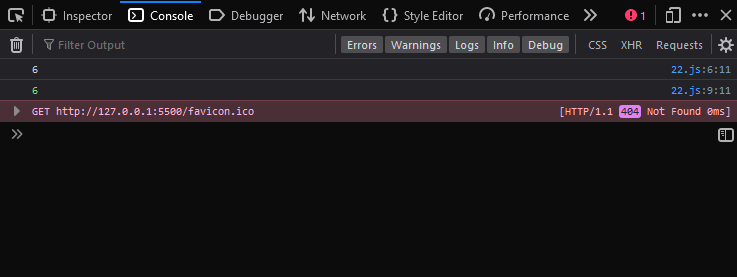
*var* n = str.search(/me in college/);

  console.log(n);

}

myFunction();

**Output :**



1. Write a program to implement backtracking- N Queen’s problems

**Code :**

*const* N = 4;

*function* printSolution(board) {

  for (*let* i = 0; i < N; i++) {

    for (*let* j = 0; j < N; j++) {

*console*.log(board[i][j], ' ');

    }

*console*.log(' ');

  }

}

*function* isSafe(*board*, *row*, *col*) {

  for (*let* i = 0; i < *col*; i++) {

    if (*board*[*row*][i] == 1) return false;

  }

  for (i = *row*, j = *col*; i >= 0 && j >= 0; i--, j--)

    if (*board*[i][j]) return false;

  for (i = *row*, j = *col*; j >= 0 && i < N; i++, j--)

    if (*board*[i][j]) return false;

  return true;

}

*function* solveNQUtil(*board*, *col*) {

  if (*col* >= N) return true;

  for (*let* i = 0; i < N; i++) {

    if (isSafe(*board*, i, *col*) == true) {

*board*[i][*col*] = 1;

      if (solveNQUtil(*board*, *col* + 1) == true) return true;

*board*[i][*col*] = 0;

    }

  }

  return false;

}

*function* solveNQ() {

*let* board = [

    [0, 0, 0, 0],

    [0, 0, 0, 0],

    [0, 0, 0, 0],

    [0, 0, 0, 0],

  ];

  if (solveNQUtil(board, 0) == false) {

    console.log('Solution does not exist');

    return false;

  }

  printSolution(board);

  return true;

}

solveNQ();

**Output :**

