**Single Linked List**

**class Node{**

**constructor(data,next=null){**

**this.data=data;**

**this.next=null;**

**}**

**}**

**class Linkedlist{**

**constructor(){**

**this.head=null;**

**}**

**// functions to be implemented**

**add(data){**

**// creates a new node**

**var node=new Node(data);**

**// to store current node**

**var temp;**

**// if list is Empty add the**

**// element and make it head**

**if (this.head==null)**

**this.head=node;**

**else{**

**temp=this.head;**

**// iterate to the end of the**

**// list**

**while(temp.next)**

**{**

**temp=temp.next;**

**}**

**// add node**

**temp.next=node;**

**}**

**}**

**addAtbeg(data){**

**var node=new Node(data);**

**var temp;**

**if (this.head==null)**

**this.head=node;**

**else{**

**//stored haed in newnode's next part**

**node.next=this.head;**

**//make head as new node**

**this.head=node;**

**}**

**}**

**addAtspe(data,index){**

**if(index>0 && index>this.size)**

**return false;**

**else{**

**//creates a new node**

**var node=new Node(data);**

**var temp,prev;**

**temp=this.head;**

**// add the element to the**

**// first index**

**if(index==0){**

**node.next=head;**

**this.head=node;**

**}**

**else{**

**temp=this.head;**

**var it=0;**

**// iterate over the list to find**

**// the position to insert**

**while(it<index){**

**it++;**

**prev=temp;**

**temp=temp.next;**

**}**

**// adding an element**

**node.next=temp;**

**prev.next=node;**

**}**

**this.size++;**

**}**

**}**

**addAtend(data){**

**var node=new Node(data);**

**var temp;**

**if (this.head==null)**

**this.head=node;**

**else{**

**temp=this.head;**

**while(temp.next)**

**{**

**temp=temp.next;**

**}**

**temp.next=node;**

**}**

**}**

**removelement(data){**

**var temp=this.head;**

**var prev=null;**

**// iterate over the list**

**while(temp!=null){**

**// comparing element with current**

**// element if found then remove the**

**// and return true**

**if(temp.data=== data){**

**if (prev==null){**

**this.head=temp.next;**

**}else{**

**prev.next=temp.next;**

**}**

**this.size--;**

**return temp.data;**

**}**

**prev=temp;**

**temp=temp.next;**

**}**

**return-1;**

**}**

**print(){**

**var temp=this.head;**

**var str=" ";**

**while(temp){**

**str +=temp.data +" ";**

**temp=temp.next;**

**}**

**console.log(str);**

**}**

**}**

**const ll=new Linkedlist()**

**ll.add(100);**

**ll.add(200);**

**ll.add(300);**

**ll.add(400);**

**ll.addAtbeg(50);**

**ll.addAtspe(250,3);**

**ll.addAtend(600);**

**//ll.removelement(250);**

**ll.print();**

**Doubly Linked List**

**class Node {**

**constructor(val) {**

**this.val = val;**

**this.next = null;**

**this.prev = null;**

**};**

**};**

**class linkedList {**

**constructor() {**

**this.head = null;**

**this.tail = null;**

**this.length = 0;**

**};**

**};**

**addHead(val) {**

**let newNode = new Node(val);**

**if (!this.head) {**

**this.head = newNode;**

**this.tail = this.head;**

**};**

**this.head.prev = newNode;**

**newNode.next = this.head;**

**this.head = newNode;**

**this.length++;**

**return this;**

**}**

**addTail(val) {**

**let newNode = new Node(val);**

**if (!this.head) {**

**this.head = newNode;**

**this.tail = newNode;**

**};**

**this.tail.next = newNode;**

**newNode.prev = this.tail;**

**this.tail = newNode;**

**this.length++;**

**return this;**

**}**

**removeHead() {**

**let removed = this.head;**

**if (!this.head) return undefined;**

**this.head = this.head.next;**

**this.head.prev = null;**

**this.length--;**

**return removed;**

**}**

**removeTail() {**

**let removed = this.tail;**

**if (!this.tail) return undefined;**

**if (this.length === 1) {**

**this.head = null;**

**this.tail = null;**

**};**

**this.tail = removed.prev;**

**this.tail.next = null;**

**this.length--;**

**return removed;**

**}**

**find(index) {**

**let current**

**if (index < 0 || index >= this.length) return undefined;**

**if (index <= this.length / 2) {**

**current = this.head;**

**for (let i = 0; i < index; i++) current = current.next;**

**} else {**

**current = this.tail;**

**for (let i = this.length; i > index; i--) current = current.prev;**

**}**

**return current;**

**}**

**insert(val, index) {**

**if (index < 0 || index > this.length) return null;**

**if (index === this.length) return this.addTail(val);**

**if (index === 0) return this.addHead(val);**

**let prev = this.find(index - 1),**

**newNode = new Node(val),**

**temp = prev.next;**

**prev.next = newNode;**

**newNode.next = temp;**

**newNode.prev = prev;**

**this.length++;**

**return true;**

**}**

**remove(index) {**

**if (index < 0 || index >= this.length) return null;**

**if (index === this.length) return this.removeTail();**

**if (index === 0) return this.removeHead();**

**let removed = this.find(index);**

**removed.prev.next = removed.next;**

**removed.next.prev = removed.prev;**

**this.length--;**

**return removed;**

**}**

**update(val, index) {**

**let node = this.find(index);**

**if (node) node.val = val;**

**return node;**

**}**

**Stack implementation using Linked list**

**class Node{**

**constructor(data,next=null){**

**this.data=data;**

**this.next=null;**

**}**

**}**

**class stackLink{**

**constructor(){**

**this.top=null;**

**}**

**push(data){**

**let node=new Node(data);**

**if(this.top){**

**node.next=this.top;**

**this.top=node;**

**}**

**else{**

**this.top=node;**

**}**

**}**

**pop(){**

**if(this.top){**

**let item=this.top;**

**this.top=this.top.next;**

**return item.data;}**

**else{**

**console.log('stack is empty');**

**}**

**}**

**print(){**

**var temp=this.top;**

**var str=" ";**

**while(temp){**

**str+=temp.data+" ";**

**temp=temp.next;**

**}**

**console.log(str);**

**}**

**}**

**const st=new stackLink();**

**st.push(100);**

**st.push(200);**

**st.print();**

**st.pop();**

**st.print();**

**Stack implementation using array**

**class Stack {**

**constructor(){**

**this.data = [];**

**this.top = 0;**

**}**

**push(element) {**

**this.data[this.top] = element;**

**this.top = this.top + 1;**

**}**

**length() {**

**return this.top;**

**}**

**peek() {**

**return this.data[this.top-1];**

**}**

**isEmpty() {**

**return this.top === 0;**

**}**

**pop() {**

**if( this.isEmpty() === false ) {**

**this.top = this.top -1;**

**return this.data.pop(); // removes the last element**

**}**

**}**

**print() {**

**var top = this.top - 1; // because top points to index where new element to be inserted**

**while(top >= 0) { // print upto 0th index**

**console.log(this.data[top]);**

**top--;**

**}**

**}**

**reverse() {**

**this.\_reverse(this.top - 1 );**

**}**

**\_reverse(index) {**

**if(index != 0) {**

**this.\_reverse(index-1);**

**}**

**console.log(this.data[index]);**

**}**

**}**

**const st=new Stack()**

**st.push(100)**

**st.push(200)**

**st.push(300)**

**st.push(400)**

**st.print()**

**st.pop()**

**st.print()**

**Queue implementation using Linked list**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**class Queue {**

**constructor() {**

**this.front= null;**

**this.rear = null;**

**this.length = 0;**

**}**

**enqueue(data) {**

**const node = new Node(data); // creates the node using class Node**

**if (this.front) { // if the first Node exitsts**

**this.rear.next = node; // inserts the created node after the tail of our Queue**

**this.rear = node; // now the created node is the last node**

**} else { // if there are no nodes in the Queue**

**this.front = node; // the created node is a head**

**this.rear = node // also the created node is a tail in Queue because it is single.**

**}**

**this.length++; // increases the length of Queue by 1**

**}**

**dequeue() {**

**const temp = this.front;**

**if(this.length===0){**

**console.log('queue is empty');**

**}else{// saves the link to the head which we need to remove**

**this.front = this.front.next; // moves the head link to the second Node in the Queue**

**this.length--; // decreaments the length of our Queue**

**return temp.data; // returns the removed Node's value**

**}**

**}**

**print() {**

**let temp = this.front; // saves a link to the head of the queue**

**while(temp) { // goes through each Node of the Queue**

**console.log(temp.data); // prints the value of the Node in console**

**temp = temp.next; // moves link to the next node after head**

**}**

**}**

**}**

**const qu=new Queue();**

**qu.enqueue(100);**

**qu.enqueue(200);**

**qu.enqueue(300);**

**qu.print();**

**qu.dequeue();**

**qu.print();**

**qu.dequeue();**

**qu.print();**

**qu.dequeue();**

**qu.print();**

**qu.dequeue();**

**qu.print();**

**Queue implementation using Array**

**class Queue {**

**constructor(){**

**this.data = [];**

**this.rear = 0;**

**this.size = 10;**

**}**

**enqueue(element) {**

**if(this.rear < this.size ) {**

**this.data[this.rear] = element;**

**this.rear = this.rear + 1;**

**}**

**}**

**length() {**

**return this.rear;**

**}**

**isEmpty() {**

**return this.rear === 0;**

**}**

**getFront() {**

**if(this.isEmpty() === false) {**

**return this.data[0];**

**}**

**}**

**getLast() {**

**if(this.isEmpty() === false) {**

**return this.data[ this.rear - 1 ] ;**

**}**

**}**

**dequeue() {**

**if(this.isEmpty() === false) {**

**this.rear = this.rear-1;**

**return this.data.shift();**

**}**

**}**

**print() {**

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

**console.log(this.data[i]);**

**}**

**}**

**clear() {**

**this.data.length = 0;**

**this.rear = 0;**

**}**

**}**

**const qu=new Queue()**

**qu.enqueue(12)**

**qu.enqueue(18)**

**qu.enqueue(20)**

**qu.print()**

**qu.dequeue()**

**qu.print()**

**Hash Table**

**class HashTable {**

**constructor() {**

**this.values = {};**

**this.length = 0;**

**this.size = 0;**

**}**

**calculateHash(key) {**

**return key.toString().length % this.size;**

**}**

**add(key, value) {**

**const hash = this.calculateHash(key);**

**If (!this.values.hasOwnProperty(hash)) {**

**this.values[hash] = {};**

**}**

**if (!this.values[hash].hasOwnProperty(key)) {**

**this.length++;**

**}**

**this.values[hash][key] = value;**

**}**

**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;**

**}**

**}**

**}**

**//create object of type hash table**

**const ht = new HashTable();**

**//add data to the hash table ht**

**ht.add("Canada", "300");**

**ht.add("Germany", "100");**

**ht.add("Italy", "50");**

**console.log(ht.search("Italy"));**

**Binary Tree**