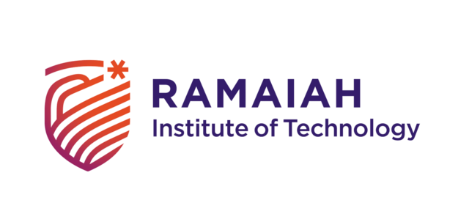
 **RAMAIAH INSTITUTE OF TECHNOLOGY**

MSRIT NAGAR, BENGALURU, 560054



**A Report on**

**LEET CODE PROGRAMS**

*Submitted in partial fulfilment of the OTHER COMPONENT requirements as a part of the Data Structures Lab subject with code ISL36 for the III Semester of degree of* ***Bachelor of Engineering in Information Science and Engineering***

**Submitted by**

**SRUSHTI S SHANKAR**

**1MS22IS137**

**Under the Guidance of**

|  |
| --- |
| **Faculty In-charge**  **MR. Shivananda S.**  **Assistant Professor**  **Dept. of ISE** |

**Department of Information Science and Engineering**

**Ramaiah Institute of Technology**

2023 - 2024

|  |  |  |
| --- | --- | --- |
| **No.** | **TOPIC** | **DIFFICULTLY LEVEL** |
| 1 | FIBONACCI NUMBER | Easy |
| 2 | VALID PARANTHESES | Easy |
| 3 | IMPLEMENT QUEUE USING STACKS | Medium |
| 4 | DESIGN CIRCULAR QUEUE | Medium |
| 5 | REVERSE LINKED LIST | Easy |
| 6 | REMOVE Nth NODE FROM END OF LIST | Medium |
| 7 | SAME TREE | Easy |
| 8 | MAXIMUM BINARY TREE | Medium |
| 9 | ODD EVEN LINKED LIST | Medium |
| 10 | BINARY TREE PREORDER TRAVERSAL | Medium |

**1)FIBONACCI NUMBER**

int fib(int n){

if(n==0)

return 0;

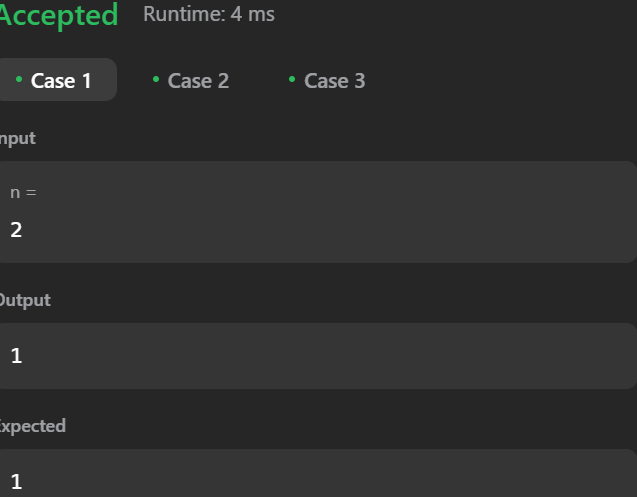
if(n==1)

return 1;

int ans = fib(n-1)+fib(n-2);

return ans;

}



**2)VALID PARANTHESES**

typedef struct {

char \*arr;

int top;

int capacity;

} Stack;

Stack\* createStack(int capacity) {

Stack \*stack = (Stack\*)malloc(sizeof(Stack));

stack->arr = (char\*)malloc(capacity \* sizeof(char));

stack->top = -1;

stack->capacity = capacity;

return stack;

}

bool isEmpty(Stack \*stack) {

return stack->top == -1;

}

void push(Stack \*stack, char value) {

stack->arr[++stack->top] = value;

}

char pop(Stack \*stack) {

if (!isEmpty(stack)) {

return stack->arr[stack->top--];

}

return '\0';

}

bool isValid(char \*s) {

int length = strlen(s);

Stack \*stack = createStack(length);

for (int i = 0; i < length; i++) {

char ch = s[i];

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

push(stack, ch);

} else {

if (isEmpty(stack)) {

return false;

}

char topChar = pop(stack);

if ((ch == ')' && topChar != '(') ||

(ch == ']' && topChar != '[') ||

(ch == '}' && topChar != '{')) {

return false;

}

}

}

bool result = isEmpty(stack);

free(stack->arr);

free(stack);

return result;

}



**3)IMPLEMENT QUEUE USING STACKS**

typedef struct {

int \*arr;

int top;

int capacity;

} Stack;

Stack\* createStack(int capacity) {

Stack\* stack = (Stack\*)malloc(sizeof(Stack));

stack->arr = (int\*)malloc(capacity \* sizeof(int));

stack->top = -1;

stack->capacity = capacity;

return stack;

}

bool isEmpty(Stack\* stack) {

return stack->top == -1;

}

void push(Stack\* stack, int item) {

stack->arr[++stack->top] = item;

}

int pop(Stack\* stack) {

if (isEmpty(stack))

return -1; // Stack underflow

return stack->arr[stack->top--];

}

typedef struct {

Stack\* s1; // For enqueue operation

Stack\* s2; // For dequeue operation

} MyQueue;

MyQueue\* myQueueCreate() {

MyQueue\* obj = (MyQueue\*)malloc(sizeof(MyQueue));

obj->s1 = createStack(1000); // Set appropriate capacity

obj->s2 = createStack(1000); // Set appropriate capacity

return obj;

}

void myQueuePush(MyQueue\* obj, int x) {

push(obj->s1, x);

}

int myQueuePop(MyQueue\* obj) {

if (isEmpty(obj->s2)) {

while (!isEmpty(obj->s1)) {

push(obj->s2, pop(obj->s1));

}

}

return pop(obj->s2);

}

int myQueuePeek(MyQueue\* obj) {

if (isEmpty(obj->s2)) {

while (!isEmpty(obj->s1)) {

push(obj->s2, pop(obj->s1));

}

}

return obj->s2->arr[obj->s2->top];

}

bool myQueueEmpty(MyQueue\* obj) {

return isEmpty(obj->s1) && isEmpty(obj->s2);

}

void myQueueFree(MyQueue\* obj) {

free(obj->s1->arr);

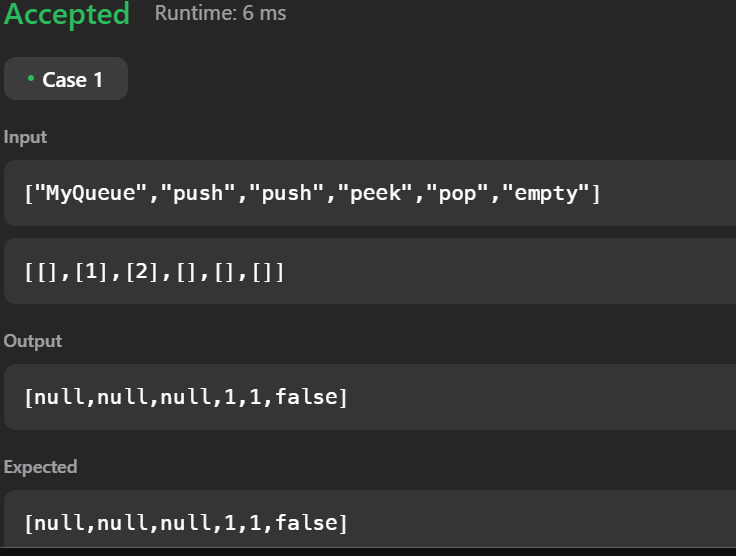
free(obj->s1);

free(obj->s2->arr);

free(obj->s2);

free(obj);

}



**4)DESIGN CIRCULAR QUEUE**

typedef struct {

int \*arr;

int front, rear, size, capacity;

} MyCircularQueue;

MyCircularQueue\* myCircularQueueCreate(int k);

bool myCircularQueueEnQueue(MyCircularQueue\* obj, int value);

bool myCircularQueueDeQueue(MyCircularQueue\* obj);

int myCircularQueueFront(MyCircularQueue\* obj);

int myCircularQueueRear(MyCircularQueue\* obj);

bool myCircularQueueIsEmpty(MyCircularQueue\* obj);

bool myCircularQueueIsFull(MyCircularQueue\* obj);

void myCircularQueueFree(MyCircularQueue\* obj);

MyCircularQueue\* myCircularQueueCreate(int k) {

MyCircularQueue\* obj = (MyCircularQueue\*)malloc(sizeof(MyCircularQueue));

obj->arr = (int\*)malloc((k + 1) \* sizeof(int));

obj->front = 0;

obj->rear = 0;

obj->size = 0;

obj->capacity = k + 1;

return obj;

}

bool myCircularQueueEnQueue(MyCircularQueue\* obj, int value) {

if (myCircularQueueIsFull(obj)) return false;

obj->arr[obj->rear] = value;

obj->rear = (obj->rear + 1) % obj->capacity;

obj->size++;

return true;

}

bool myCircularQueueDeQueue(MyCircularQueue\* obj) {

if (myCircularQueueIsEmpty(obj)) return false;

obj->front = (obj->front + 1) % obj->capacity;

obj->size--;

return true;

}

int myCircularQueueFront(MyCircularQueue\* obj) {

if (myCircularQueueIsEmpty(obj)) return -1;

return obj->arr[obj->front];

}

int myCircularQueueRear(MyCircularQueue\* obj) {

if (myCircularQueueIsEmpty(obj)) return -1;

return obj->arr[(obj->rear - 1 + obj->capacity) % obj->capacity];

}

bool myCircularQueueIsEmpty(MyCircularQueue\* obj) {

return obj->size == 0;

}

bool myCircularQueueIsFull(MyCircularQueue\* obj) {

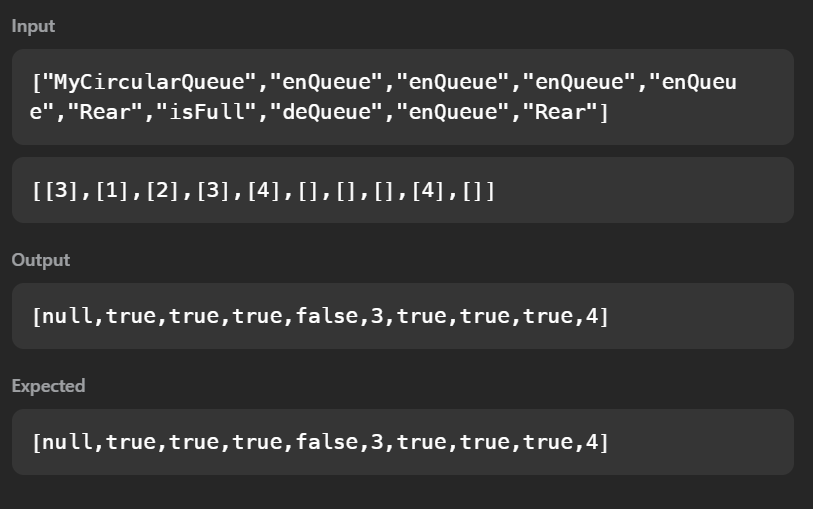
return obj->size == obj->capacity - 1;

}

void myCircularQueueFree(MyCircularQueue\* obj) {

free(obj->arr);

free(obj);



**5)REVERSE LINKED LIST**

struct ListNode {

int val;

struct ListNode \*next;

};

struct ListNode\* reverseList(struct ListNode\* head) {

struct ListNode \*prev = NULL, \*current = head, \*nextNode = NULL;

while (current != NULL) {

nextNode = current->next; // Store the next node

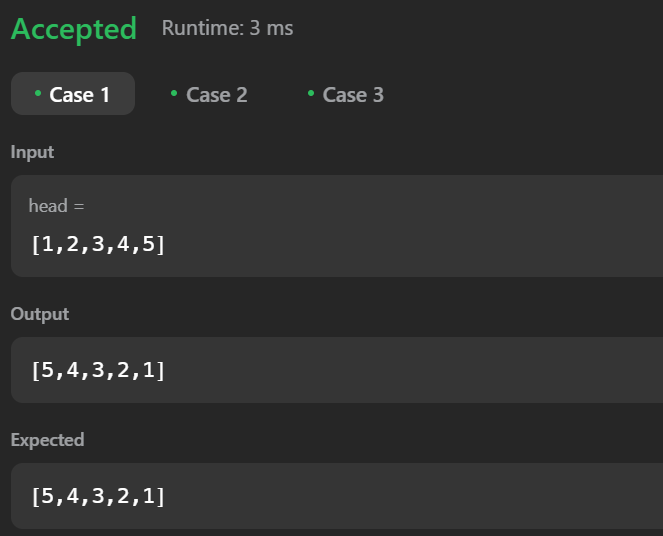
current->next = prev; // Reverse the link

prev = current;

current = nextNode;

return prev;

}



**6)REMOVE Nth NODE FROM END OF LIST**

struct ListNode\* removeNthFromEnd(struct ListNode\* head, int n) {

if(head==NULL || (head->next==NULL)){

return NULL;

}

int i=0,a;

struct ListNode \*temp,\*d;

temp=head;

while(temp!=NULL){

i++;

temp=temp->next;

}

if(n==i)

return head->next;

temp=head;

a=i-n-1;

while(a!=0){

temp=temp->next;

a--;

}

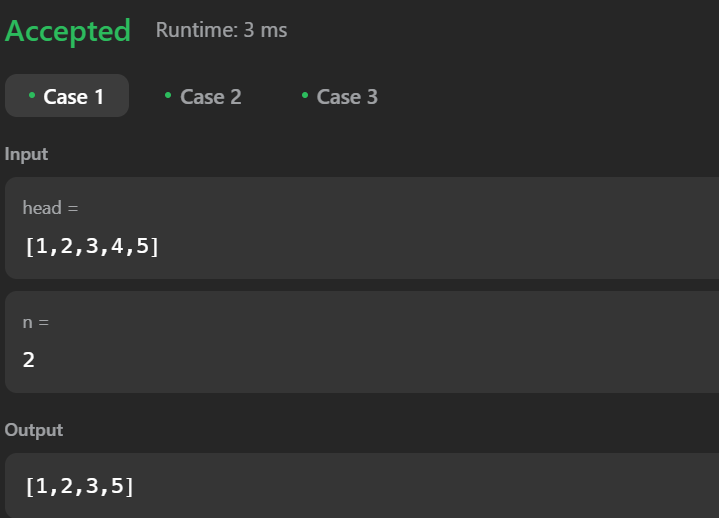
d=temp->next;

temp->next=temp->next->next;

free(d);

return head;

}



**7)SAME TREE**

bool isSameTree(struct TreeNode\* p, struct TreeNode\* q) {

if(!p && !q) return true;

if(!p || !q) return false;

bool l=false,r=false;

if(p->val==q->val)

{

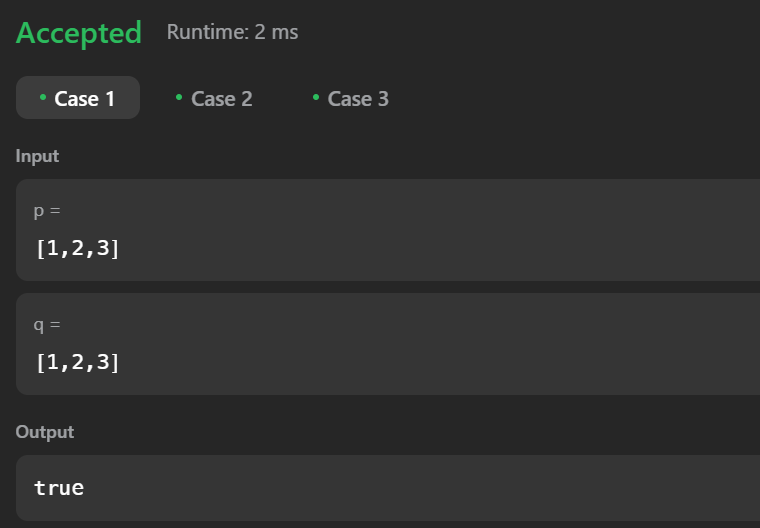
l= isSameTree(p->left,q->left);

r=isSameTree(p->right,q->right);

}

return l&&r;

}



**8)MAXIMUM BINARY TREE**

typedef struct TreeNode Tree;

Tree \* constructMaximumBinaryTree(const int \* const nums, const int numsLen){

if (0 == numsLen){

return NULL;

}

int maxIdx = 0;

for (int i = 1; i < numsLen; i += 1){

if (nums[i] > nums[maxIdx]){

maxIdx = i;

}

}

Tree \* const pTree = (Tree \*)malloc( sizeof (Tree) );

pTree->val = nums[maxIdx];

pTree->left = constructMaximumBinaryTree(nums, maxIdx);

pTree->right = constructMaximumBinaryTree( nums + (maxIdx + 1), numsLen - (maxIdx + 1) );

return pTree;

}



**9) ODD EVEN LINKED LIST**

struct ListNode\* oddEvenList(struct ListNode\* head) {

if (!head || !head->next) {

return head; // Handle empty or single node list

}

struct ListNode \*odd = head, \*even = head->next, \*evenHead = even;

while (even && even->next) {

odd->next = even->next;

odd = odd->next;

even->next = odd->next;

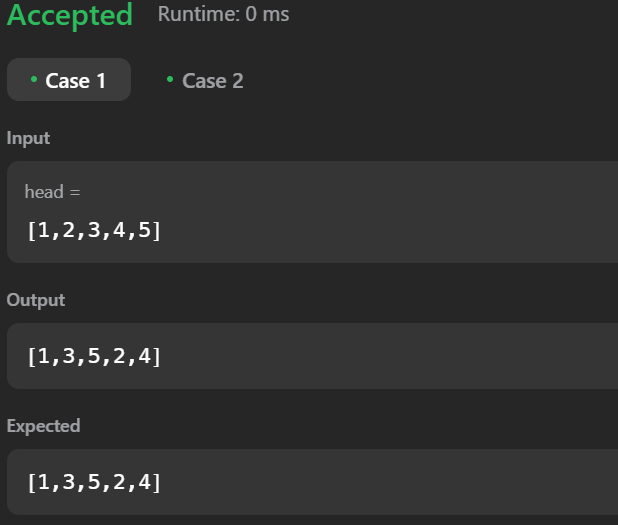
even = even->next;

}

odd->next = evenHead;

return head;

}



**10) BINARY TREE PREORDER TRAVERSAL**

#define SIZE 1000

void preorder(struct TreeNode\* root, int\* result, int\* returnSize) {

if (root == NULL) return;

result[(\*returnSize)++] = root->val;

preorder(root->left, result, returnSize);

preorder(root->right, result, returnSize);

}

int\* preorderTraversal(struct TreeNode\* root, int\* returnSize) {

int\* result = (int\*)malloc(SIZE \* sizeof(int));

\*returnSize = 0;

preorder(root, result, returnSize);

return result;

}

