RAMAIAH INSTITUTE OF TECHNOLOGY

MSRIT NAGAR, BENGALURU,560054

LeetCode

[Data Structures Lab]

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

Submitted by

Candidate Name

1MS22IS139

SUHAS S

Under the Guidance of

Faculty Incharge

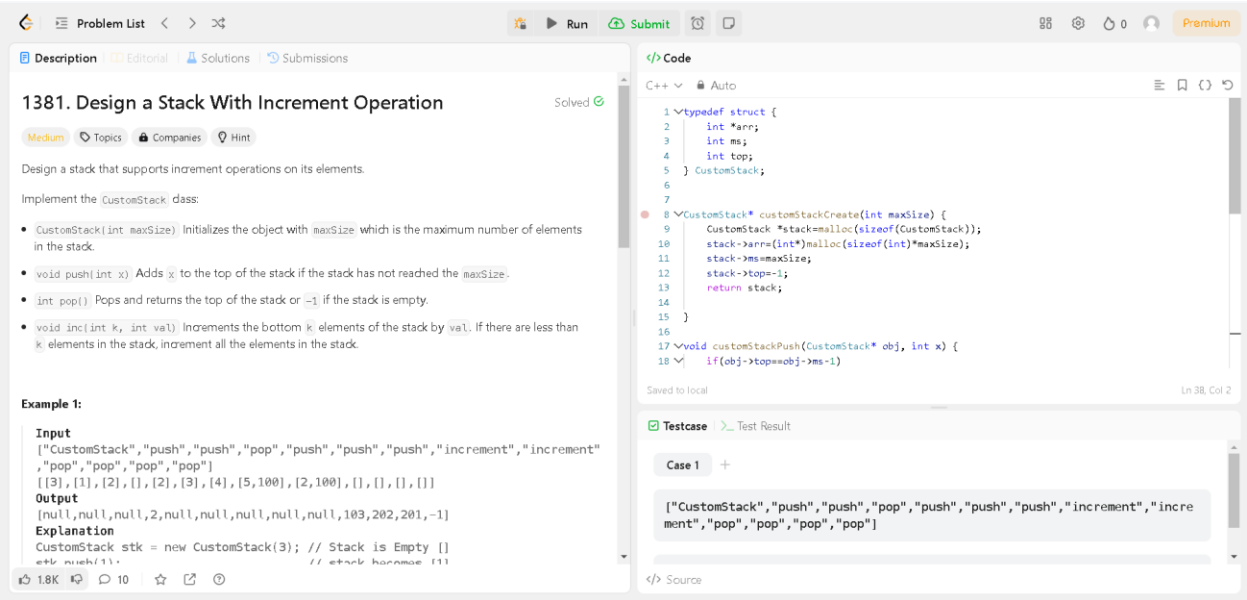
SHIVANANDA S

Department of Information Science and Engineering

Ramaiah Institute of Technology

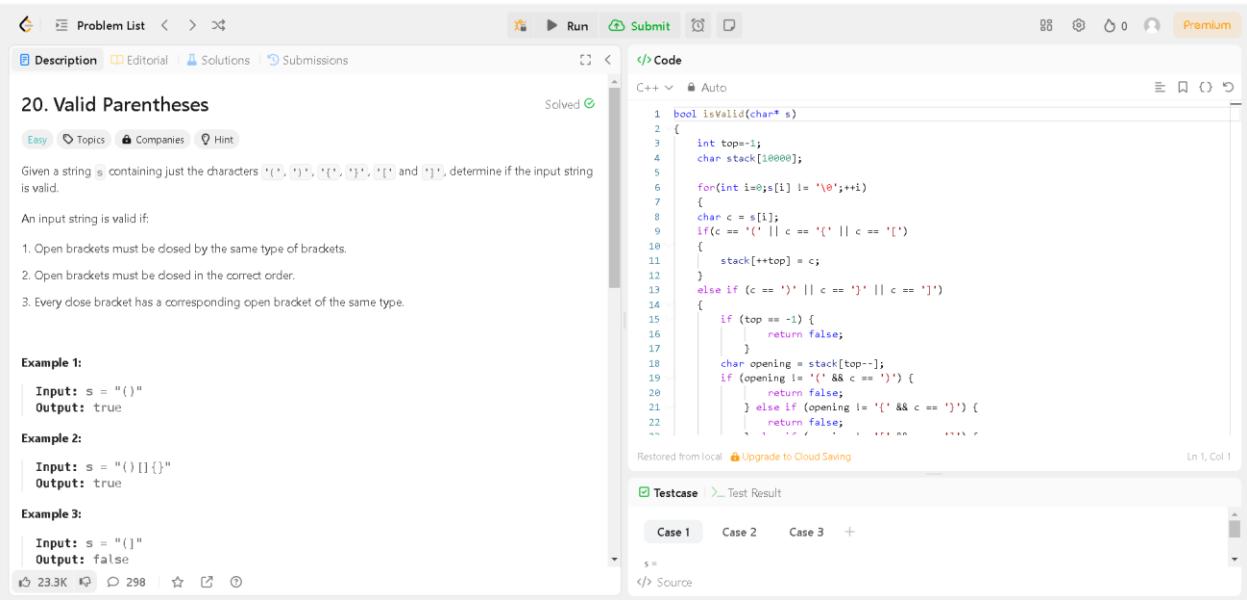
2023 – 2024

1. Design a Stack With Increment Operator



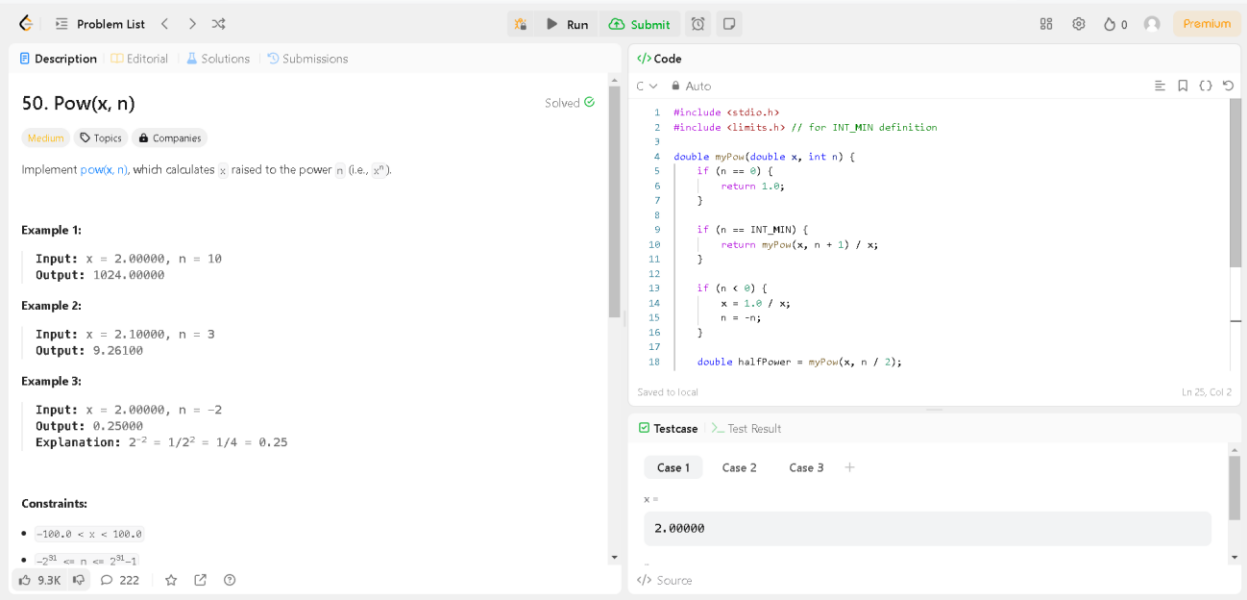

```c
typedef struct {
    int *arr;
    int ms;
    int top;
} CustomStack;


CustomStack* customStackCreate(int maxSize) {
    CustomStack *stack=malloc(sizeof(CustomStack));
    stack->arr=(int*)malloc(sizeof(int)*maxSize);
    stack->ms=maxSize;
    stack->top=-1;
    return stack;

}

void customStackPush(CustomStack* obj, int x) {
    if(obj->top==obj->ms-1)
        return;
    obj->arr[++(obj->top)]=x;
}

int customStackPop(CustomStack* obj) {
    if(obj->top==-1)
        return -1;
    return obj->arr[(obj->top)--];
}

void customStackIncrement(CustomStack* obj, int k, int val) {
    if(k>=obj->ms)
        k=obj->ms;
    for(int i=0;i<k;i++)
        obj->arr[i]+=val;
}

void customStackFree(CustomStack* obj) {

}
```

2. Valid Parentheses



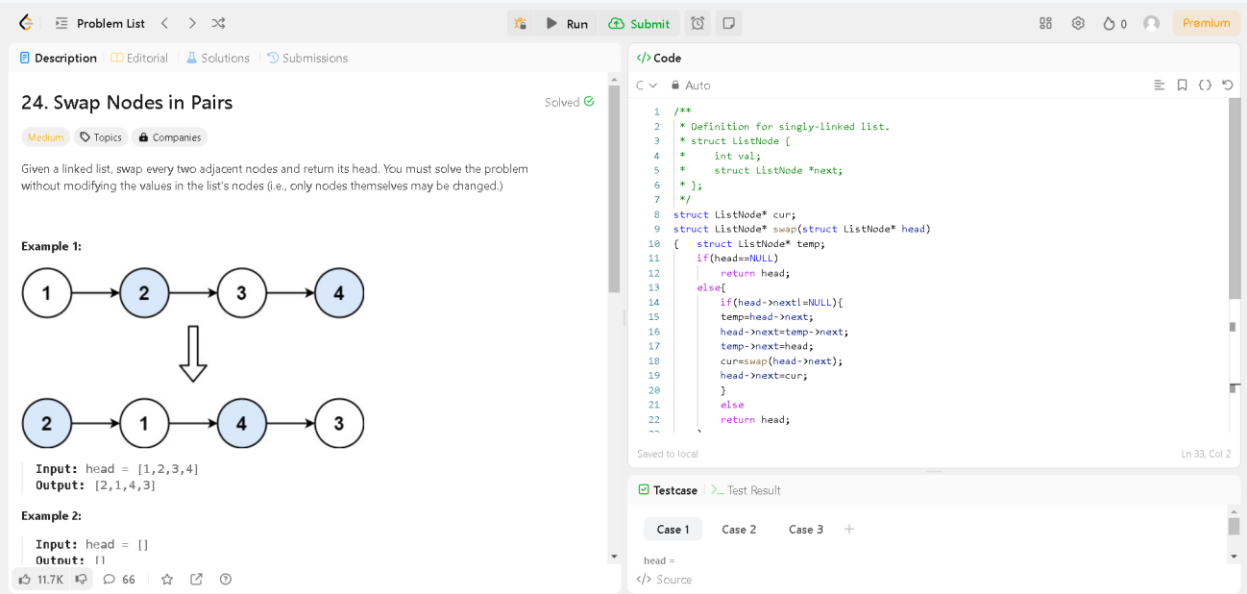

```c
bool isValid(char* s)
{
    int top=-1;
    char stack[10000];

    for(int i=0;s[i] != '\0';++i)
    {
    char c = s[i];
    if(c == '(' || c == '{' || c == '[')
    {
        stack[++top] = c;
    }
    else if (c == ')' || c == '}' || c == ']')
    {
        if (top == -1) {
                return false;
            }
        char opening = stack[top--];
        if (opening != '(' && c == ')') {
                return false;
            } else if (opening != '{' && c == '}') {
                return false;
            } else if (opening != '[' && c == ']') {
                return false;
            }
        }
    }
    return top == -1;
}
```

3. Pow(x,n)



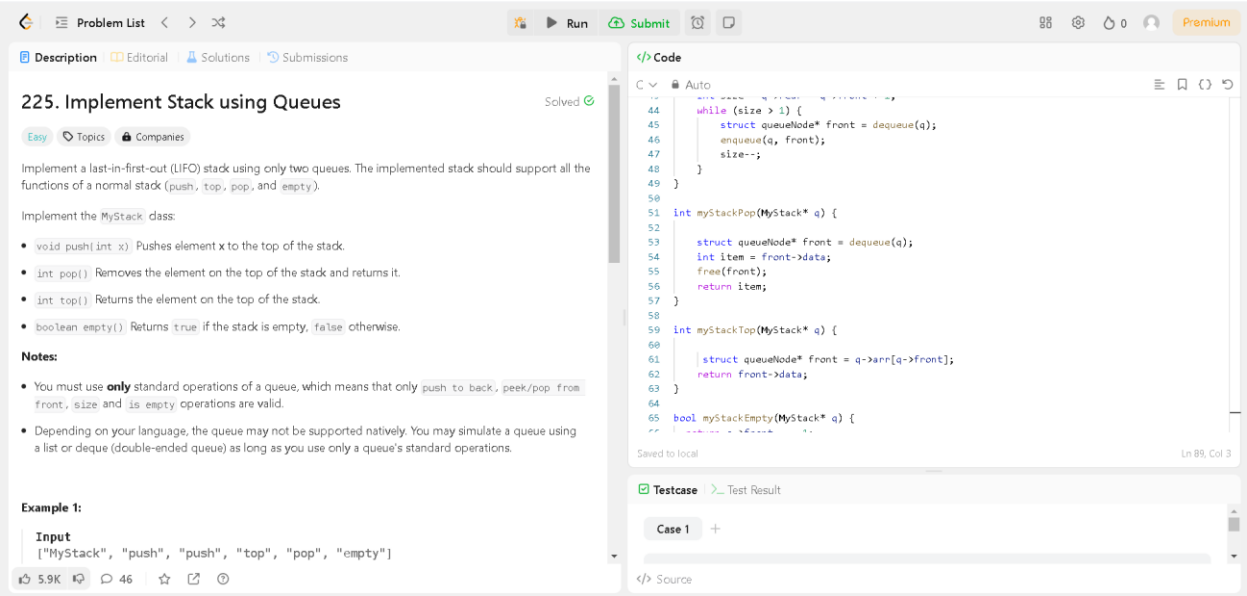

```c
#include <stdio.h>
#include <limits.h> // for INT_MIN definition

double myPow(double x, int n) {
    if (n == 0) {
        return 1.0;
    }

    if (n == INT_MIN) {
        return myPow(x, n + 1) / x;
    }

    if (n < 0) {
        x = 1.0 / x;
        n = -n;
    }

    double halfPower = myPow(x, n / 2);

    if (n % 2 == 0) {
        return halfPower * halfPower;
    } else {
        return halfPower * halfPower * x;
    }
}
```

4. Swap Nodes In Paris




```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* cur;
struct ListNode* swap(struct ListNode* head)
{   struct ListNode* temp;
    if(head==NULL)
        return head;
    else{
        if(head->next!=NULL){
        temp=head->next;
        head->next=temp->next;
        temp->next=head;
        cur=swap(head->next);
        head->next=cur;
        }
        else
        return head;
    }
    return temp;
}
struct ListNode* swapPairs(struct ListNode* head) {
    if(head==NULL || head->next==NULL)
        return head;

    head=swap(head);
    return head;

}
```

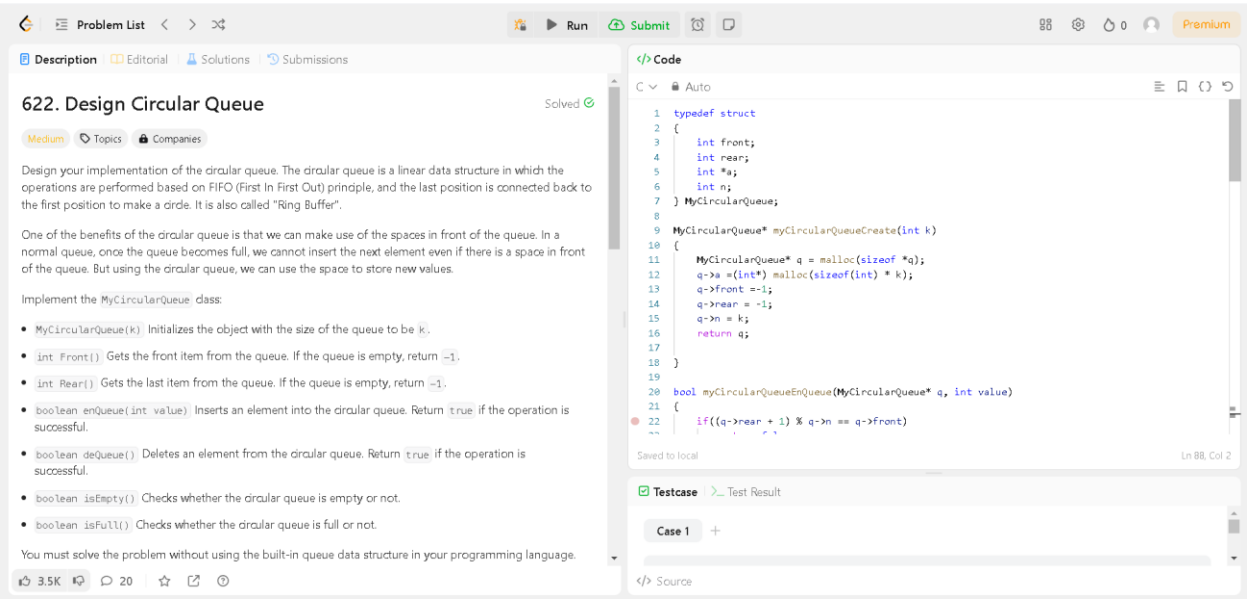
5. Implement Stack Using Queues




```c
struct queueNode{
    int data;
};

typedef struct {
    struct queueNode *arr[101];
    int front;
    int rear;
} MyStack;


MyStack* myStackCreate() {
    MyStack *q =(MyStack *)malloc(sizeof(MyStack));
    q->front=-1;
    q->rear=-1;
    return q;
}

void enqueue(MyStack* q, struct queueNode* item) {
    q->arr[++q->rear] = item;
    if (q->front == -1) {
        q->front = 0;
    }
}

struct queueNode* dequeue(MyStack* q) {

    struct queueNode* item = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return item;
}
```

```c
void myStackPush(MyStack* q, int x) {

  struct queueNode* node = (struct queueNode*) malloc(sizeof(struct queueNode));
    node->data = x;
    enqueue(q, node);
    int size = q->rear - q->front + 1;
    while (size > 1) {
        struct queueNode* front = dequeue(q);
        enqueue(q, front);
        size--;
    }
}

int myStackPop(MyStack* q) {

    struct queueNode* front = dequeue(q);
    int item = front->data;
    free(front);
    return item;
}

int myStackTop(MyStack* q) {

     struct queueNode* front = q->arr[q->front];
    return front->data;
}

bool myStackEmpty(MyStack* q) {
  return q->front == -1;
}

void myStackFree(MyStack* q) {
     while (!myStackEmpty(q)) {
        struct queueNode* front = dequeue(q);
        free(front);
    }
    free(q);
}
```

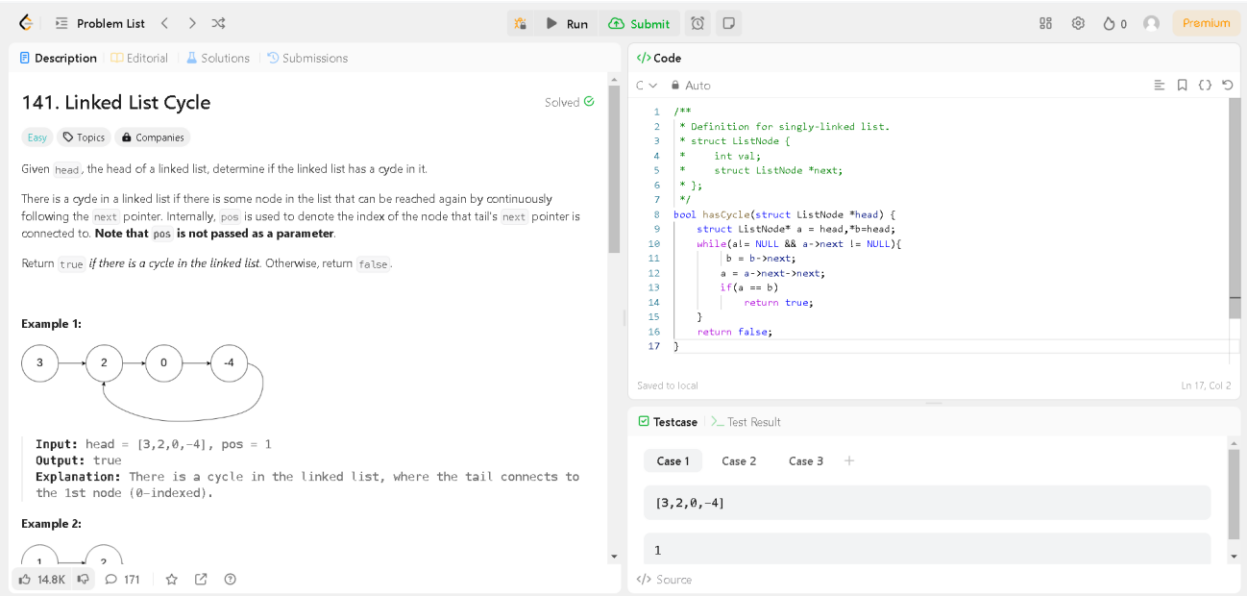
6. Design Circular Queue




```c
typedef struct
{
    int front;
    int rear;
    int *a;
    int n;
} MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* q = malloc(sizeof *q);
    q->a =(int*) malloc(sizeof(int) * k);
    q->front =-1;
    q->rear = -1;
    q->n = k;
    return q;

}

bool myCircularQueueEnQueue(MyCircularQueue* q, int value)
{
    if((q->rear + 1) % q->n == q->front)
        return false;
    else
    {
    if (q->front == -1)
        q->front = 0;
    q->rear=(q->rear+1)%q->n;
    q->a[q->rear]=value;
    return true;
    }

}

bool myCircularQueueDeQueue(MyCircularQueue* q)
{
    if(q->front==-1)
        return false;
    else
    {

    if (q->front == q->rear)
    {
      q->front = -1;
      q->rear = -1;
    }
    else
      q->front = (q->front + 1) % q->n;
        return true;
    }

}
```

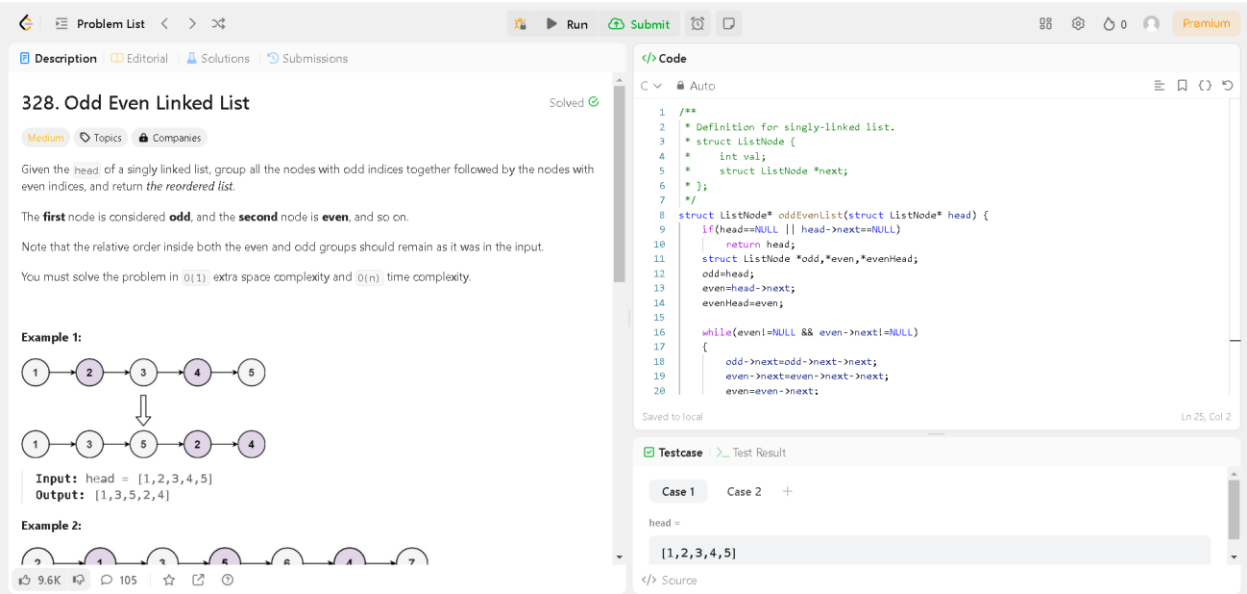
```c
int myCircularQueueFront(MyCircularQueue* q)
{
    if(q->front==-1)
        return -1;
    return q->a[q->front];

}

int myCircularQueueRear(MyCircularQueue* q)
{
    if(q->front==-1)
        return -1;

        return (q->a[q->rear]);

}

bool myCircularQueueIsEmpty(MyCircularQueue* q)
{
    if(q->front==-1)
        return true;
    return false;
}

bool myCircularQueueIsFull(MyCircularQueue* q)
{
    if((q->rear + 1) % q->n == q->front)
        return true;
  return false;
}

void myCircularQueueFree(MyCircularQueue* obj)
{
    free(obj);
}
```

7. Linked List Cycle



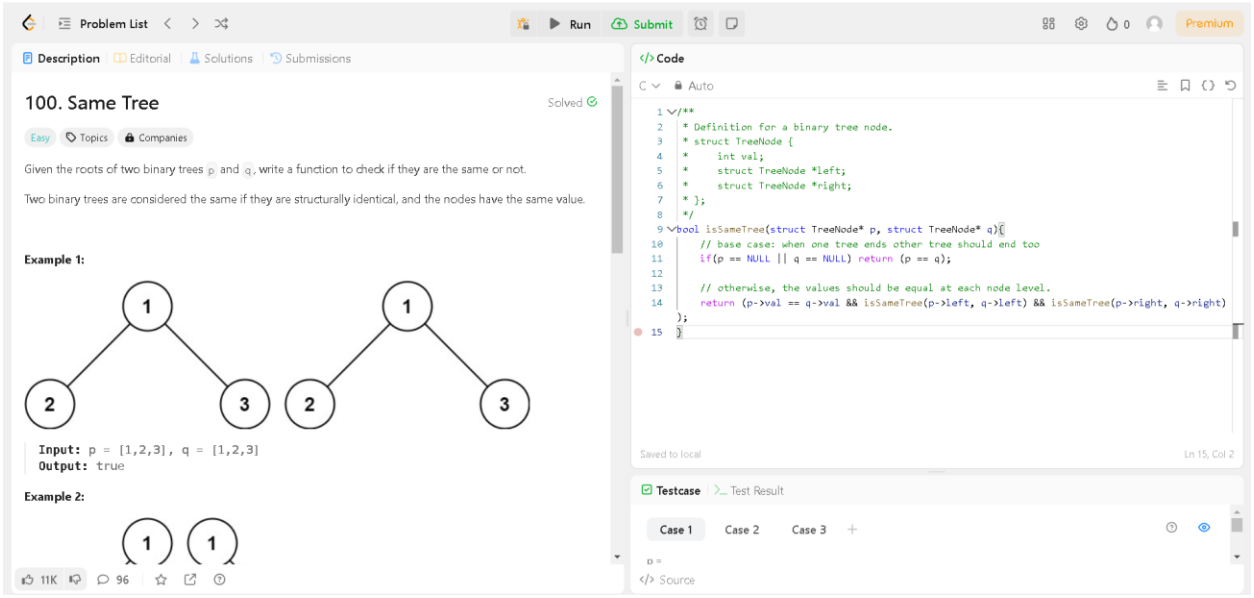

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
bool hasCycle(struct ListNode *head) {
    struct ListNode* a = head,*b=head;
    while(a!= NULL && a->next != NULL){
         b = b->next;
        a = a->next->next;
        if(a == b)
            return true;
    }
    return false;
}
```

8. Odd Even Linked List



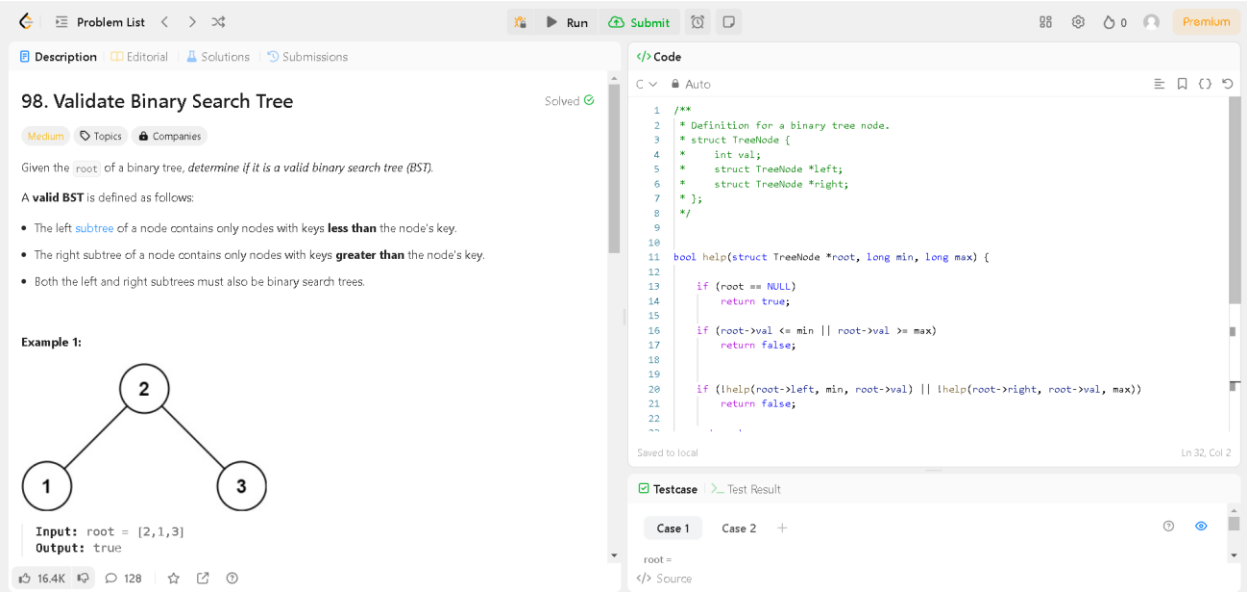

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* oddEvenList(struct ListNode* head) {
    if(head==NULL || head->next==NULL)
        return head;
    struct ListNode *odd,*even,*evenHead;
    odd=head;
    even=head->next;
    evenHead=even;

    while(even!=NULL && even->next!=NULL)
    {
        odd->next=odd->next->next;
        even->next=even->next->next;
        even=even->next;
        odd=odd->next;
    }
    odd->next=evenHead;
    return head;
}
```

9. Same Tree




```c
bool isSameTree(struct TreeNode* p, struct TreeNode* q){
    // base case: when one tree ends other tree should end too
    if(p == NULL || q == NULL) return (p == q);

    // otherwise, the values should be equal at each node level.
    return (p->val == q->val && isSameTree(p->left, q->left) && isSameTree(p->right, q->right));
}
```

10. Validate Binary Search Tree




```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */


bool help(struct TreeNode *root, long min, long max) {

    if (root == NULL)
        return true;

    if (root->val <= min || root->val >= max)
        return false;


    if (!help(root->left, min, root->val) || !help(root->right, root->val, max))
        return false;

    return true;
}

bool isValidBST(struct TreeNode* root) {

    if (root == NULL || (root->left == NULL && root->right == NULL))
        return true;

    return help(root, LONG_MIN, LONG_MAX);
}
```

REAL TIME PROBLEM: Restaurant Menu Manager

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct MenuItem {
    char name[50];
    float price;
    struct MenuItem* next;
};

void displayMenu(struct MenuItem* head) {
    printf("Menu:\n");
    printf("-----------------------------\n");
    while (head != NULL) {
        printf("%s - $%.2f\n", head->name, head->price);
        head = head->next;
    }
    printf("-----------------------------\n");
}

struct MenuItem* addItem(struct MenuItem* head, char name[], float price)
{
    struct MenuItem* newItem = (struct MenuItem*)malloc(sizeof(struct
MenuItem));
    if (newItem == NULL) {
        printf("Error: Memory allocation failed.\n");
        return head;
    }

    strncpy(newItem->name, name, sizeof(newItem->name));
    newItem->price = price;
    newItem->next = head;
    printf("Item added: %s - $%.2f\n", newItem->name, newItem->price);

    return newItem;
}

struct MenuItem* deleteItem(struct MenuItem* head, char name[]) {
    struct MenuItem* current = head;
```

```c
    struct MenuItem* prev = NULL;

    while (current != NULL && strcmp(current->name, name) != 0) {
        prev = current;
        current = current->next;
    }

    if (current == NULL) {
        printf("Error: Item not found.\n");
        return head;
    }

    if (prev == NULL) {
        head = current->next;
    } else {
        prev->next = current->next;
    }

    printf("Item deleted: %s - $%.2f\n", current->name, current->price);
    free(current);

    return head;
}

void freeMenu(struct MenuItem* head) {
    struct MenuItem* current = head;
    while (current != NULL) {
        struct MenuItem* next = current->next;
        free(current);
        current = next;
    }
}

float calculateTotal(struct MenuItem* order) {
    float total = 0.0;
    while (order != NULL) {
        total += order->price;
        order = order->next;
    }
    return total;
}

int main() {
```

```c
    struct MenuItem* menu = NULL;
    struct MenuItem* order = NULL;

    menu = addItem(menu, "Burger", 5.99);
    menu = addItem(menu, "Pizza", 8.49);
    menu = addItem(menu, "Salad", 4.99);

    displayMenu(menu);

    order = addItem(order, "Burger", 5.99);
    order = addItem(order, "Pizza", 8.49);

    printf("\nOrder Summary:\n");
    displayMenu(order);

    float totalBill = calculateTotal(order);
    printf("\nTotal Bill: $%.2f\n", totalBill);

    // Cleaning up memory
    freeMenu(menu);
    freeMenu(order);

    return 0;
}
```

Output:

```
PS D:\Myfiles\Msrit\Notes\Sem-3\data_structures\Leetcode programs> gcc Restauran_Menu_Manager.c
PS D:\Myfiles\Msrit\Notes\Sem-3\data_structures\Leetcode programs> ./a.exe
Item added: Burger - $5.99
Item added: Pizza - $8.49
Item added: Salad - $4.99
Menu:
------------------------------
Salad - $4.99
Pizza - $8.49
Burger - $5.99
------------------------------
Item added: Burger - $5.99
Item added: Pizza - $8.49

Order Summary:
Menu:
------------------------------
Pizza - $8.49
Burger - $5.99
------------------------------

Total Bill: $14.48
PS D:\Myfiles\Msrit\Notes\Sem-3\data_structures\Leetcode programs>
```