# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

**LAB REPORT**
**on**

# Data Structures using C

***Submitted by***

**S Rakhal**
**1BM22CS229**

***in partial fulfillment for the award of the degree of***
**BACHELOR OF ENGINEERING**
***in***
**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**
**(Autonomous Institution under VTU)**
**BENGALURU-560019**
**December-2023 to April-2023**
**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

# CERTIFICATE

This is to certify that the Lab work entitled "**Data Structures using C**" carried out by**S Rakhal(1BM22CS229),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester December-2023 to March-2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **Data Structures using C (23CS3PCDST)** work prescribed for the said degree.

Lakshmi Neelima
Assistant professor

Dr. Jyothi S Nayak
Professor and
Head of Department
Of CSE BMSCE

**Index Sheet**



**Course outcomes:**

| | |
|---|---|
| CO1 | Apply the concept of linear and nonlinear data structures. |
| CO2 | Analyze data structure operations for a given problem |
| CO3 | Design and develop solutions using the operations of linear and nonlinear data structure for a given specification. |
| CO4 | Conduct practical experiments for demonstrating the operations of different data structures. |

**Lab program 1:**

**Write a program to simulate the working of stack using an array with the following:**
**a) Push**
**b) Pop**
**c) Display**
**The program should print appropriate messages for stack overflow, stack underflow.**

**Code:**

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

#define size 10
int pos = -1;
int stack[size];

void push(int a);
int pop();
void display();

int main(){
    printf("1. Push\n2. Pop\n3. Display stack\n4. Exit\nEnter
choice: ");
    int choice;
    int a;
    scanf("%d", &choice);
    while(choice != 4){
        switch(choice){
            case 1:
                printf("Enter integer to be pushed: ");
                scanf("%d", &a);
                push(a);
                break;
            case 2:
                a = pop();
                printf("Integer poppped = %d\n", a);
                break;
            case 3:
                display();
                break;
            default:
                printf("Invalid input");
                break;
```

```
        }
        printf("Enter choice: ");
        scanf("%d", &choice);
    }

}

void push(int a){
    if (pos == 9){
        printf("Stack Overflow condition");
        return;
    }
    stack[++pos] = a;
}

int pop(){
    if (pos == -1){
        printf("Stack Underflow condition");
        return (int) NULL;
    }
    return stack[pos--];
}

void display(){
    for(int i = 0; i < size; i++){
        printf("%d ", stack[i]);
    }
    printf("\n");
}
```

**Output:**

```
1. Push
2. Pop
3. Display stack
4. Exit
Enter choice: 1
Enter integer to be pushed: 3
Enter choice: 1
Enter integer to be pushed: 4
Enter choice: 2
Integer poppped = 4
Enter choice: 1
Enter integer to be pushed: 5
Enter choice: 3
3 5 0 0 0 0 0 0 0 0
Enter choice: 4
```

**Lab program 2:**

**WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)**

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

#define size 20

void push(char a);
char pop();
void display();
char* postfix(char* exp);
char* prefix(char* exp);
bool character(char c);
bool lower_precedence(char op1, char op2);
bool isEmpty();


int pos = -1;
char stack[size];
int n;
```

```
int main(){

    printf("Enter size of expression in terms of characters: ");
    scanf("%d", &n);
    fflush(stdin);
    char* expr = (char*) malloc(size*sizeof(char));
    printf("Enter infix expression: ");
    scanf("%[^\n]s", expr);
    char* postfixexp = postfix(expr);
    printf("Postfix expression: %s\n", postfixexp);
    char* prefixexp = prefix(expr);
    printf("Prefix expression: %s", prefixexp);

    return 0;
}

bool isEmpty(){
    return pos == -1;
}

void push(char a){
    if (pos == size-1){
        printf("Stack Overflow condition");
        return;
    }
    stack[++pos] = a;
}

char pop(){
    if (pos == -1){
        printf("Stack Underflow condition");
        return (char) NULL;
    }

    char return_value = stack[pos];
    stack[pos] = (char) NULL;
    pos--;
    return return_value;
}

void display(){
    printf("Stack: ");
```

```
    for(int i = 0; i < size; i++){
        printf("%c ", stack[i]);
    }
    printf("\n");
}

bool character(char c){
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c
>= '0' && c <= '9');
}

bool lower_precedence(char op1, char op2){
    if (op1 == op2 && op2 == '^') return false;
    char op_order[] = {'^', '/', '*', '+', '-'};
    int o1, o2;
    for(int i = 0; i < 5; i++){
        if (op_order[i] == op1) o1 = i;
        if (op_order[i] == op2) o2 = i;
    }

    return o1 <= o2;

}

char* postfix(char* exp){
    char* return_exp = (char*) malloc((size+3)*sizeof(char));
    int current = 0;
    for(int i = 0; i < n; i++){

        if (character(exp[i])){
            return_exp[current++] = exp[i];
        }
        else if(exp[i] == '+' || exp[i] == '-' || exp[i] == '*' ||
exp[i] == '/' || exp[i] == '^'){
            if (isEmpty()) push(exp[i]);

            else if(lower_precedence(stack[pos], exp[i])){
                while(lower_precedence(stack[pos], exp[i]) &&
!isEmpty()){
                    if (stack[pos] != '(') return_exp[current++] =
pop();
                    else{
```

```
                    pos--;
                    break;
                }
            }
            push(exp[i]);
        }
        else push(exp[i]);
    }
    else if(exp[i] == '('){
        push('(');
    }
    else if(exp[i] == ')'){
        while(stack[pos] != '(' && !isEmpty()){
            return_exp[current++] = pop();
        }
    }


}

while(!isEmpty()){
    if (stack[pos] != '(') return_exp[current++] = pop();
    else pos--;
}

return return_exp;

}

char* prefix(char* exp){
    char* buffer = malloc(size*sizeof(char));
    for(int i = n-1; i >= 0; i--){
        buffer[n-i-1] = exp[i];
        if (buffer[n-i-1] == '(') buffer[n-i-1] = ')';
        else if (buffer[n-i-1] == ')') buffer[n-i-1] = '(';
    }

    printf("Reversed String: %s\n", buffer);
    char* return_exp = malloc(size*sizeof(char));
    int current = 0;
    for(int i = 0; i < n; i++){
```

```
        if (character(buffer[i])){
            return_exp[current++] = buffer[i];
        }
        else if(buffer[i] == '+' || buffer[i] == '-' || buffer[i]
== '*' || buffer[i] == '/' || buffer[i] == '^'){
            if (isEmpty()) push(buffer[i]);

            else if(lower_precedence(stack[pos], buffer[i])){
                while(lower_precedence(stack[pos], buffer[i]) &&
!isEmpty()){
                    if (stack[pos] != '(') return_exp[current++] =
pop();
                    else{
                        pos--;
                        break;
                    }
                }
                push(buffer[i]);
            }
            else push(buffer[i]);
        }
        else if(buffer[i] == '('){
            push('(');
        }
        else if(buffer[i] == ')'){
            while(stack[pos] != '(' && !isEmpty()){
                return_exp[current++] = pop();
            }
        }


    }



    while(!isEmpty()){
        if (stack[pos] != '(') return_exp[current++] = pop();
        else pos--;
    }

    char* final = (char*) malloc(size*sizeof(char));
    for(int i = 0; i < strlen(return_exp); i++)
```

```
        final[i] = return_exp[strlen(return_exp)-i-1];

    return final;
}
```

**Output:**

```
Enter size of expression in terms of characters: 9
Enter infix expression: a*b+c*d-e
Postfix expression: ab*cd*+e-
```

**Lab 3:**

**3a)**

**WAP to simulate the working of a queue of integers using an array. Provide the following operations: Insert, Delete, Display. The program should print appropriate messages for queue empty and queue overflow conditions.**

**Code:**

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

#define size 5

void push(int a);
int pop();
void display();

int fpos = -1, rpos = -1;
int queue[size];

int main(){
    int choice;
    printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter
choice: ");
    scanf("%d", &choice);
    int a;
    while(choice != 4){
        switch(choice){
            case 1:
                printf("Enter integer to be pushed: ");
                scanf("%d", &a);
                push(a);
```

```
                break;
            case 2:
                a = pop();
                printf("Popped integer = %d\n", a);
                break;
            case 3:
                display();
                break;
            default:
                printf("Idk");
                break;
        }
        printf("Enter choice: ");
        scanf("%d", &choice);
    }

}

void push(int a){
    if (fpos == -1 && rpos == -1){
        queue[++rpos] = a;
        fpos++;
        return;
    }
    else if (rpos == size-1){
        printf("Queue overflow condition\n");
        return;
    }
    else{
        queue[++rpos] = a; return;
    }
}

int pop(){
    if (fpos == -1){
        printf("Queue Underflow condition\n");
    }
    int n = queue[fpos];
    queue[fpos] = (int) NULL;
    fpos++;
    return n;
}
```

```
void display(){
    printf("Queue: ");
    for(int i = 0; i < size; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
```

**Output:**

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 1
Enter integer to be pushed: 2
Enter choice: 1
Enter integer to be pushed: 3
Enter choice: 1
Enter integer to be pushed: 4
Enter choice: 1
Enter integer to be pushed: 5
Enter choice: 1
Enter integer to be pushed: 6
Enter choice: 3
Queue: 2 3 4 5 6
Enter choice: 2
Popped integer = 2
Enter choice: 2
Popped integer = 3
Enter choice: 3
Queue: 0 0 4 5 6
Enter choice: 1
Enter integer to be pushed: 7
Queue overflow condition
Enter choice: 4

Process returned 0 (0x0)   execution time : 28.952 s
```

**3b ) WAP to simulate the working of a circular queue of integers using an array. Provide the following operations: Insert, Delete & Display.The program should print appropriate messages for queue empty and queue overflow conditions**

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```c
#include <string.h>

#define size 5

void push(int a);
int pop();
void display();

int fpos = -1, rpos = -1;
int queue[size];

int main(){
    int choice;
    printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter
choice: ");
    scanf("%d", &choice);
    int a;
    while(choice != 4){
        switch(choice){
            case 1:
                printf("Enter integer to be pushed: ");
                scanf("%d", &a);
                push(a);
                // printf("fpos = %d; rpos = %d\n", fpos%size,
rpos%size);
                break;
            case 2:
                a = pop();
                printf("Popped integer = %d\n", a);
                // printf("fpos = %d; rpos = %d\n", fpos%size,
rpos%size);
                break;
            case 3:
                display();
                break;
            default:
                printf("Idk");
                break;
        }
        printf("Enter choice: ");
        scanf("%d", &choice);
    }
```

```
}

void push(int a){
    if (fpos == -1 && rpos == -1){
        queue[++rpos] = a;
        fpos++;
        return;
    }
    else if ((rpos+1)%size == (fpos%size)){
        printf("Queue overflow condition\n");
        return;
    }
    else{
        rpos++;
        queue[(rpos%size)] = a;
        return;
    }
}

int pop(){
    if (fpos == -1){
        printf("Queue Underflow condition\n");
    }
    int n = queue[fpos%size];
    queue[fpos%size] = (int) NULL;
    fpos++;
    return n;
}

void display(){
    printf("Queue: ");
    for(int i = 0; i < size; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
```

**Output:**

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 1
Enter integer to be pushed: 2
Enter choice: 1
Enter integer to be pushed: 3
Enter choice: 1
Enter integer to be pushed: 4
Enter choice: 1
Enter integer to be pushed: 5
Enter choice:
1
Enter integer to be pushed: 6
Enter choice: 3
Queue: 2 3 4 5 6
Enter choice: 2
Popped integer = 2
Enter choice: 3
Queue: 0 3 4 5 6
Enter choice: 1
Enter integer to be pushed: 1
Enter choice: 3
Queue: 1 3 4 5 6
Enter choice: 2
Popped integer = 3
Enter choice: 3
Queue: 1 0 4 5 6
Enter choice: 4

Process returned 0 (0x0)   execution time : 32.409 s
```

**Lab 4:**

**WAP to Implement Singly Linked List with following operations**

**a) Create a linked list.**

**b) Insertion of a node at first position, at any position and at end of list.**

**Display the contents of the linked list.**

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
```

```
    int data;
    struct Node *next;
} node;

node* head = NULL;
int count = 0;

void insert(int data, int position);
void display();

int main(){
    int data, choice, pos;
    printf("1. Insert\n2. Delete\n3. Exit\nChoice: ");
    scanf("%d", &choice);
    while(choice != 2){
        if (choice == 1){
            printf("Enter data and position: ");
            scanf("%d%d", &data, &pos);
            insert(data, pos);
            printf("Count: %d\n", count);
        }
        display();
        printf("Enter choice: ");
        scanf("%d", &choice);
    }


    return 0;
}

void insert(int data, int position){
    if (position == 0){
        node* new_node = (node*)malloc(sizeof(node));
        new_node->data = data;
        new_node->next = NULL;
        head = new_node;
        count++;
        return;

    } else if (position == count){
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
```

```
        new_node->next = NULL;
        node* temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = new_node;
        count++;
        return;

    } else if (position > count || position < 0){
        printf("Unable to insert at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
            temp = temp->next;
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = temp->next;
        temp->next = new_node;
        count++;
        return;
    }
}
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Choice: 1
Enter data and position: 1 0
Count: 1
Linked List: 1
Enter choice: 1
Enter data and position: 2 1
Count: 2
Linked List: 1 2
Enter choice: 1
Enter data and position: 3 2
Count: 3
Linked List: 1 2 3
Enter choice: 1
Enter data and position: 4 1
Count: 4
Linked List: 1 4 2 3
Enter choice: 1
Enter data and position: 5 2
Count: 5
Linked List: 1 4 5 2 3
Enter choice: 3

Process returned 0 (0x0)   execution time : 47.266 s
Press any key to continue.
```

**Program - Leetcode platform:**

**Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.**

**Implement the MinStack class:**

**MinStack() initializes the stack object.**

**void push(int val) pushes the element val onto the stack.**

**void pop() removes the element on the top of the stack.**

**int top() gets the top element of the stack.**

**int getMin() retrieves the minimum element in the stack.**

**You must implement a solution with O(1) time complexity for each function.**

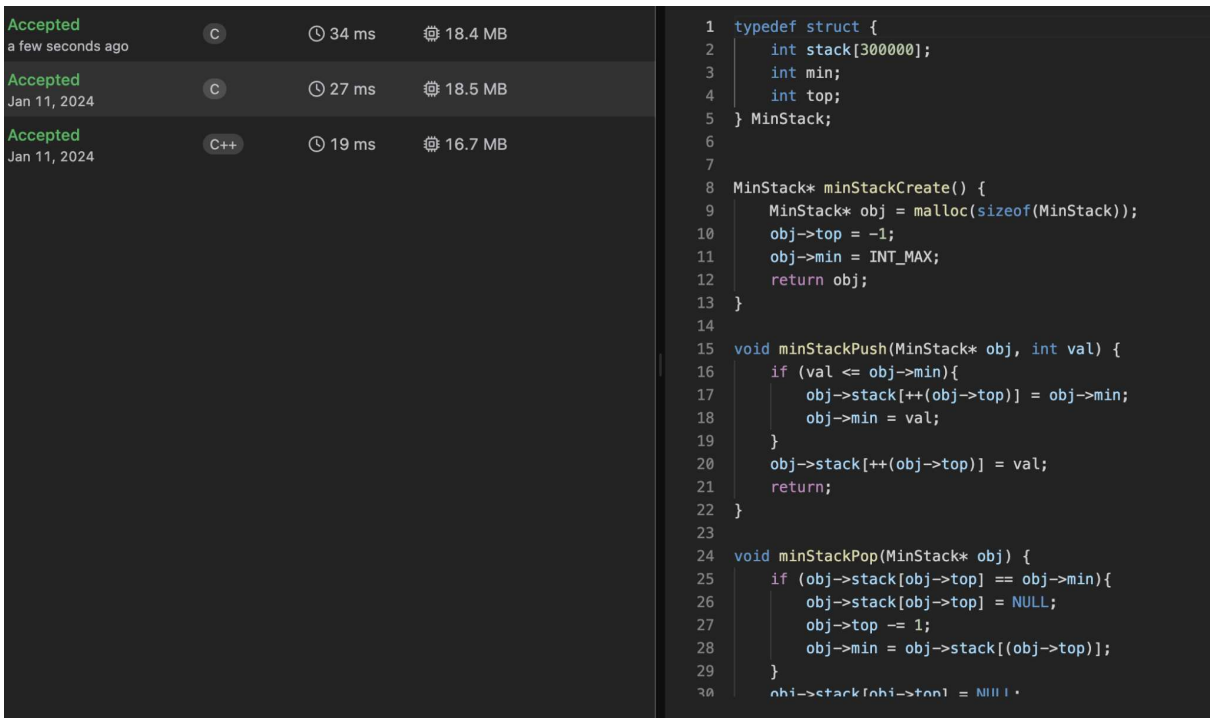
**Code:**

```
typedef struct {
```

```
    int stack[300000];
    int min;
    int top;
} MinStack;


MinStack* minStackCreate() {
    MinStack* obj = malloc(sizeof(MinStack));
    obj->top = -1;
    obj->min = INT_MAX;
    return obj;
}

void minStackPush(MinStack* obj, int val) {
    if (val <= obj->min){
        obj->stack[++(obj->top)] = obj->min;
        obj->min = val;
    }
    obj->stack[++(obj->top)] = val;
    return;
}

void minStackPop(MinStack* obj) {
    if (obj->stack[obj->top] == obj->min){
        obj->stack[obj->top] = NULL;
        obj->top -= 1;
        obj->min = obj->stack[(obj->top)];
    }
    obj->stack[obj->top] = NULL;
    obj->top -= 1;
}

int minStackTop(MinStack* obj) {
    return obj->stack[obj->top];
}

int minStackGetMin(MinStack* obj) {
    return obj->min;
}

void minStackFree(MinStack* obj) {
    free(obj);
```

```
}
```

**Output:**




**Lab 5:**

**WAP to Implement Singly Linked List with following operations**

**a) Create a linked list.**

**b) Deletion of first element, specified element and last element in the list.**

**c) Display the contents of the linked list.**

**Code:**

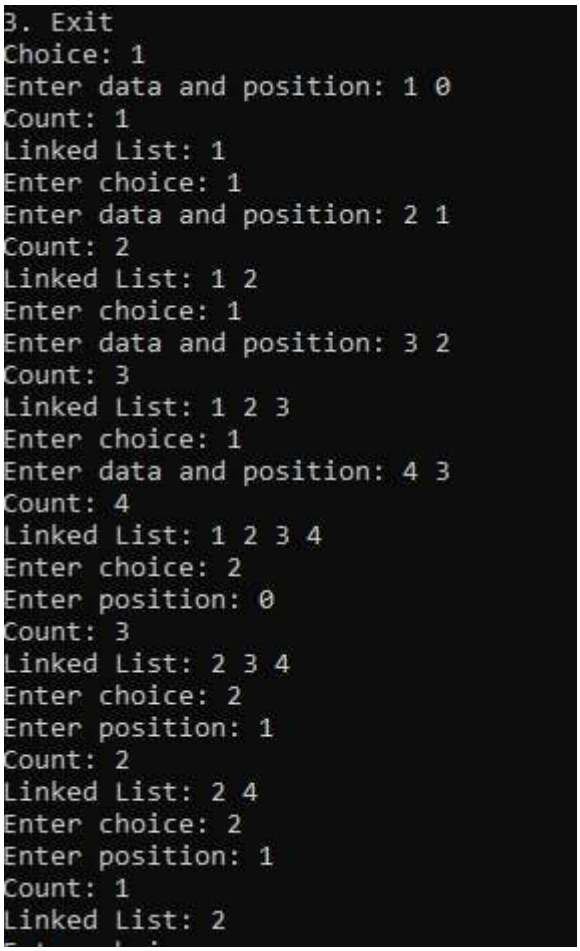
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
```

```
    int data;
    struct Node *next;
} node;

node* head = NULL;
int count = 0;

void insert(int data, int position);
void delete(int position);
void display();

int main(){
    int data, choice, pos;
    printf("1. Insert\n2. Delete\n3. Exit\nChoice: ");
    scanf("%d", &choice);
    while(choice != 3){
        if (choice == 1){
            printf("Enter data and position: ");
            scanf("%d%d", &data, &pos);
            insert(data, pos);
            printf("Count: %d\n", count);
        } else if (choice == 2){
            printf("Enter position: ");
            scanf("%d", &pos);
            delete(pos);
            printf("Count: %d\n", count);
        }
        display();
        printf("Enter choice: ");
        scanf("%d", &choice);
    }


    return 0;
}

void insert(int data, int position){
    if (position == 0){
        node* new_node = (node*)malloc(sizeof(node));
        new_node->data = data;
        new_node->next = head;
        head = new_node;
```

```
        count++;
        return;

    } else if (position == count){
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = NULL;
        node* temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = new_node;
        count++;
        return;

    } else if (position > count || position < 0){
        printf("Unable to insert at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
            temp = temp->next;
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = temp->next;
        temp->next = new_node;
        count++;
        return;
    }
}

void delete(int position){
    if (position == 0){
        node* temp = head;
        head = head->next;
        free(temp);
        count--;
        return;
    } else if (position == count-1){
        node* temp = head;
        for(int i = 1; i < count-1; i++)
            temp = temp->next;
        node* temp1 = temp->next;
```

```
        temp->next = NULL;
        free(temp1);
        count--;
        return;
    } else if (position > count || position < 0){
        printf("Unable to delete at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
            temp = temp->next;
        node* temp1 = temp->next;
        temp->next = temp1->next;
        free(temp1);
        count--;
        return;
    }
}

void display(){
    node* temp = head;
    printf("Linked List: ");
    while (temp->next != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
    printf("\n");
}
```

**Output:**




**Program - Leetcode platform**

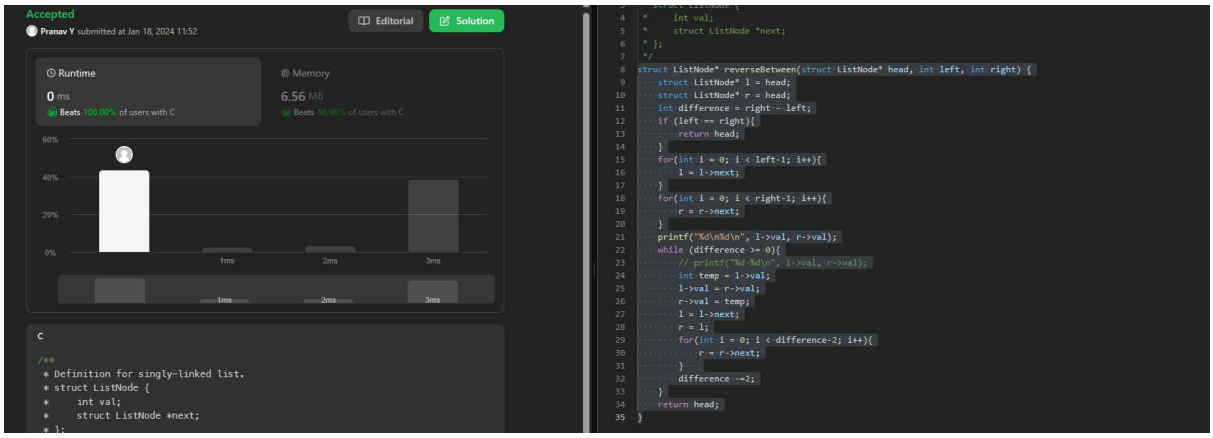
**Given the head of a singly linked list, reverse the list, and return *the reversed list*.**

## Code:

```
struct ListNode* reverseBetween(struct ListNode* head, int left,
int right) {
    struct ListNode* l = head;
    struct ListNode* r = head;
    int difference = right - left;
    if (left == right){
        return head;
    }
    for(int i = 0; i < left-1; i++){
```

```
        l = l->next;
    }
    for(int i = 0; i < right-1; i++){
        r = r->next;
    }
    printf("%d\n%d\n", l->val, r->val);
    while (difference >= 0){
        // printf("%d %d\n", l->val, r->val);
        int temp = l->val;
        l->val = r->val;
        r->val = temp;
        l = l->next;
        r = l;
        for(int i = 0; i < difference-2; i++){
            r = r->next;
        }
        difference -=2;
    }
    return head;
}
```

## Output:



**Lab 6**

**a) WAP to Implement Single Link List with following operations:**

- **Sort the linked list**
- **Reverse the linked list**
- **Concatenation of two linked lists.**

## Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
    int data;
    struct Node *next;
} node;

node *head = NULL;
node *head1 = NULL;
int count = 0;

void insert(int data, int position);
void delete(int position);
void display();
void sort();
void reverse();
void concat(node** head1, node** head2);

int main(){
    insert(2, 0);
    insert(1, 1);
    insert(4, 2);
    insert(3, 3);
    insert(5, 4);
    printf("Original Linked List: \n");
    display();
    sort();
    printf("Sorted Linked List: \n");
    display();
    reverse();
```

```
    printf("Reversed Linked List: \n");
    display();
    head1 = head;
    head = NULL;
    insert(3, 0);
    insert(4, 1);
    insert(1, 2);
    display();
    concat(&head1, &head);
    head = head1;
    printf("Concatenating with the above linked list gives: \n");
    display();

    return 0;
}

void insert(int data, int position){
    if (position == 0){
        node* new_node = (node*)malloc(sizeof(node));
        new_node->data = data;
        new_node->next = head;
        head = new_node;
        count++;
        return;

    } else if (position == count){
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = NULL;
        node* temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = new_node;
        count++;
        return;

    } else if (position > count || position < 0){
        printf("Unable to insert at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
```

```
            temp = temp->next;
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = temp->next;
        temp->next = new_node;
        count++;
        return;
    }
}

void delete(int position){
    if (position == 0){
        node* temp = head;
        head = head->next;
        free(temp);
        count--;
        return;
    } else if (position == count-1){
        node* temp = head;
        for(int i = 1; i < count-1; i++)
            temp = temp->next;
        node* temp1 = temp->next;
        temp->next = NULL;
        free(temp1);
        count--;
        return;
    } else if (position > count || position < 0){
        printf("Unable to delete at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
            temp = temp->next;
        node* temp1 = temp->next;
        temp->next = temp1->next;
        free(temp1);
        count--;
        return;
    }
}

void sort(){
```

```
    int i, j, min_index;
    node *i_node=head, *j_node=head, *min_node=NULL;
    for(int i = 0; i < count-1; i++, i_node=i_node->next){
        // printf("Got here\n");
        min_index = i;
        min_node = i_node;
        j_node = i_node->next;
        for(int j = i+1; j < count; j++, j_node=j_node->next){
            // printf("Got here too\n");
            if (j_node->data < i_node->data){
                min_index = j;
                min_node = j_node;
            }
        }
        // printf("%d\n", min_index);
        if (min_index != i){
            // printf("Found a min element\n");
            int temp = i_node->data;
            i_node->data = min_node->data;
            min_node->data = temp;
            // display();
        }
    }
}

void reverse(){
    node *prev = NULL, *next=NULL;
    while(head != NULL){
        next = head->next;
        head->next = prev;
        prev = head;
        head = next;
    }
    head = prev;
}

void concat(node **head1, node **head2){
    node *temp1 = *head1;
    while (temp1->next != NULL){
        temp1 = temp1->next;
    }
    // printf("Got here atleast?\n");
```

```
    temp1->next = *head2;
    // printf("Got here?\n");
}

void display(){
    node* temp = head;
    printf("Linked List: ");
    while (temp->next != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
    printf("\n");
}
```

**Output:**

```
Original Linked List:
Linked List: 2 1 4 3 5
Sorted Linked List:
Linked List: 1 2 3 4 5
Reversed Linked List:
Linked List: 5 4 3 2 1
Linked List: 3 4 1
Concatenating with the above linked list gives:
Linked List: 5 4 3 2 1 3 4 1

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

**6b) WAP to Implement Single Link List to simulate Stack & Queue Operations.**

**i) Stack**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
    int data;
    struct Node *next;
} node;

node* head = NULL;
int count = 0;
```

```
void insert(int data);
int delete();
void display();

int main(){
    int data, choice, pos;
    printf("1. Insert\n2. Delete\n3. Exit\nChoice: ");
    scanf("%d", &choice);
    while(choice != 3){
        if (choice == 1){
            printf("Enter data: ");
            scanf("%d", &data);
            insert(data);
            printf("Count: %d\n", count);
        } else if (choice == 2){
            printf("Integer popped = %d\n", delete());
            printf("Count: %d\n", count);
        }
        display();
        printf("Enter choice: ");
        scanf("%d", &choice);
    }


    return 0;
}

void insert(int data){
    node* new_node = (node*)malloc(sizeof(node));
    new_node->data = data;
    new_node->next = head;
    head = new_node;
    count++;
    return;

}

int delete(){
    node* temp = head;
    head = head->next;
    int t = temp->data;
```

```
    free(temp);
    count--;
    return t;

}

void display(){
    node* temp = head;
    printf("Stack: ");
    while (temp->next != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
    printf("\n");
}
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Choice: 1
Enter data: 3
Count: 1
Linked List: 3
Enter choice: 1
Enter data: 2
Count: 2
Linked List: 2 3
Enter choice: 1
Enter data: 5
Count: 3
Linked List: 5 2 3
Enter choice: 2
Integer popped = 5
Count: 2
Linked List: 2 3
Enter choice: 2
Integer popped = 2
Count: 1
Linked List: 3
Enter choice: 3

Process returned 0 (0x0)   execution time : 14.281 s
Press any key to continue.
```

**ii)Queue**

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

typedef struct Node{
    int data;
    struct Node *next;
} node;

node* head = NULL;
int count = 0;

void insert(int data);
int delete();
void display();

int main(){
    int data, choice, pos;
    printf("1. Insert\n2. Delete\n3. Exit\nChoice: ");
    scanf("%d", &choice);
    while(choice != 3){
        if (choice == 1){
            printf("Enter data: ");
            scanf("%d", &data);
            insert(data);
            printf("Count: %d\n", count);
        } else if (choice == 2){
            printf("Integer popped = %d\n", delete());
            printf("Count: %d\n", count);
        }
        display();
        printf("Enter choice: ");
        scanf("%d", &choice);
    }


    return 0;
}

void insert(int data){
    node* new_node = malloc(sizeof(node));
```

```
    new_node->data = data;
    new_node->next = NULL;
    if (head == NULL){
        head = new_node;
        count++;
        return;
    }
    node* temp = head;
    while(temp->next != NULL)
        temp = temp->next;
    temp->next = new_node;
    count++;
    return;

}

int delete(){
    node* temp = head;
    head = head->next;
    int t = temp->data;
    free(temp);
    count--;
    return t;

}

void display(){
    node* temp = head;
    printf("Queue: ");
    while (temp->next != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
    printf("\n");
}
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Choice: 1
Enter data: 1
Count: 1
Queue: 1
Enter choice: 1
Enter data: 2
Count: 2
Queue: 1 2
Enter choice: 1
Enter data: 3
Count: 3
Queue: 1 2 3
Enter choice: 2
Integer popped = 1
Count: 2
Queue: 2 3
Enter choice: 2
Integer popped = 2
Count: 1
Queue: 3
Enter choice: 3

Process returned 0 (0x0)   execution time : 8.781 s
Press any key to continue.
```

**Lab 7**
**WAP to Implement doubly link list with primitive operations**

**a) Create a doubly linked list.**
**b) Insert a new node to the left of the node.**
**c) Delete the node based on a specific value**
**d) Display the contents of the list**

**Code:**

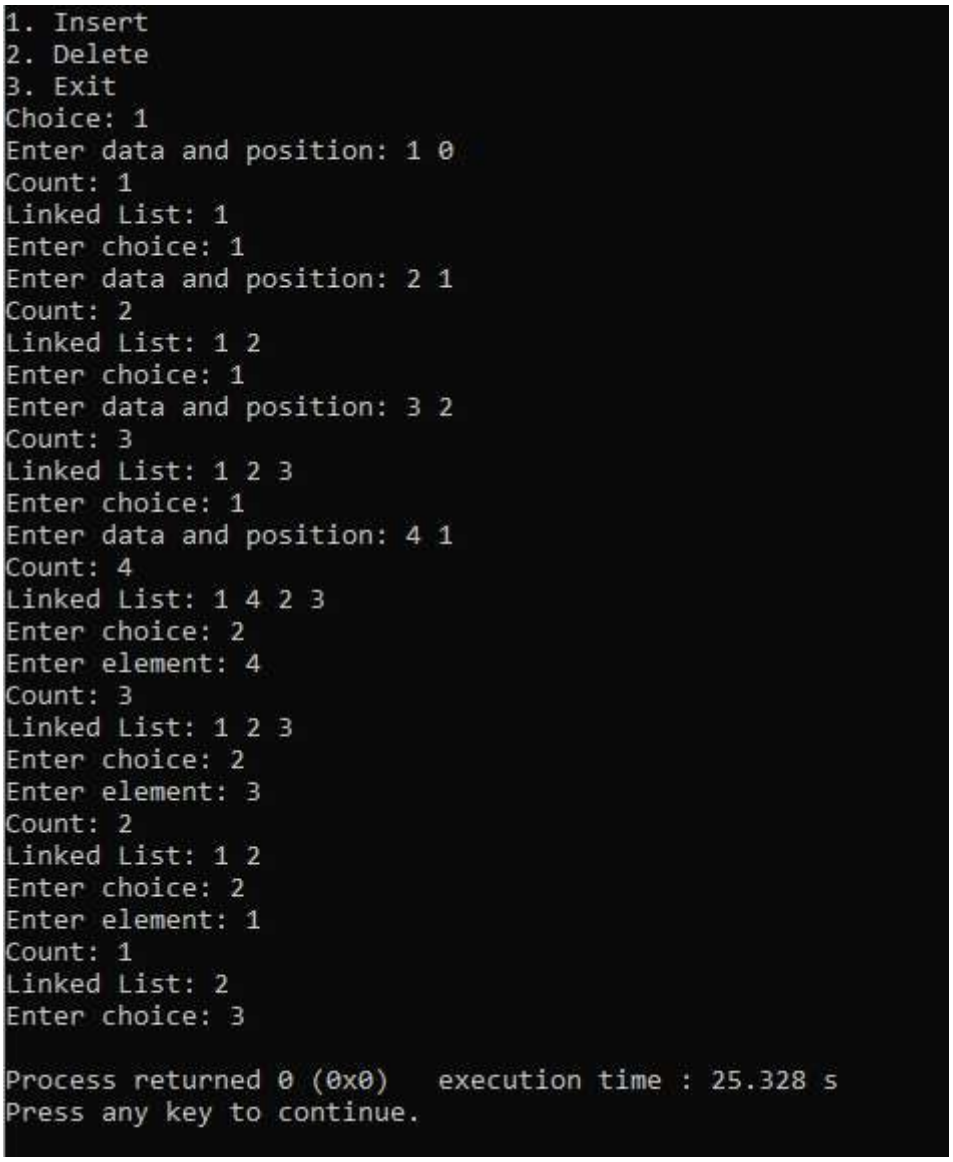
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
    int data;
    struct Node *next;
    struct Node *prev;
} node;
```

```
node* head = NULL;
int count = 0;

void insert(int data, int position);
void delete(int element);
void display();

int main(){
    int data, choice, pos;
    printf("1. Insert\n2. Delete\n3. Exit\nChoice: ");
    scanf("%d", &choice);
    while(choice != 3){
        if (choice == 1){
            printf("Enter data and position: ");
            scanf("%d%d", &data, &pos);
            insert(data, pos);
            printf("Count: %d\n", count);
        } else if (choice == 2){
            printf("Enter element: ");
            scanf("%d", &pos);
            delete(pos);
            printf("Count: %d\n", count);
        }
        display();
        printf("Enter choice: ");
        scanf("%d", &choice);
    }


    return 0;
}

void insert(int data, int position){
    if (position == 0){
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = head;
        new_node->prev = NULL;
        if (head != NULL) head->prev = new_node;
        head = new_node;
        count++;
        return;
```

```
    } else if (position == count){
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = NULL;
        node* temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = new_node;
        new_node->prev = temp;
        count++;
        return;

    } else if (position > count || position < 0){
        printf("Unable to insert at given position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position-1; i++)
            temp = temp->next;
        node* new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->next = temp->next;
        new_node->prev = temp;
        temp->next->prev = new_node;
        temp->next = new_node;
        count++;
        return;
    }
}

void delete(int element){
    int position = 0; node *temp = head;
    if (head == NULL){
        printf("List is empty, cannot delete"); return;
    }
    for(;position < count; temp=temp->next, position++)
        if (temp->data == element) break;
    if (temp == NULL){
        printf("Element does not exist in list"); return;
    }
    if (position == 0){
        node* temp = head;
```

```
        temp = temp->next;
        temp->prev = NULL;
        free(head);
        head = temp;
        count--;
        return;
    } else if (position == count-1){
        node* temp = head;
        for(int i = 1; i < count-1; i++)
            temp = temp->next;
        node* temp1 = temp->next;
        temp->next = NULL;
        free(temp1);
        count--;
        return;
    } else if (position > count || position < 0){
        printf("Unable to delete at position\n");
        return;
    } else {
        node* temp = head;
        for(int i = 0; i < position; i++)
            temp = temp->next;
        temp->next->prev = temp->prev;
        temp->prev->next = temp->next;
        free(temp);
        count--;
        return;
    }
}

void display(){
    node* temp = head;
    printf("Linked List: ");
    while (temp->next != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
    printf("\n");
}
```

**Output:**




**Leetcode**

**Given the head of a singly linked list and an integer k, split the linked list into k consecutive linked list parts.**

**The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.**

**The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.**

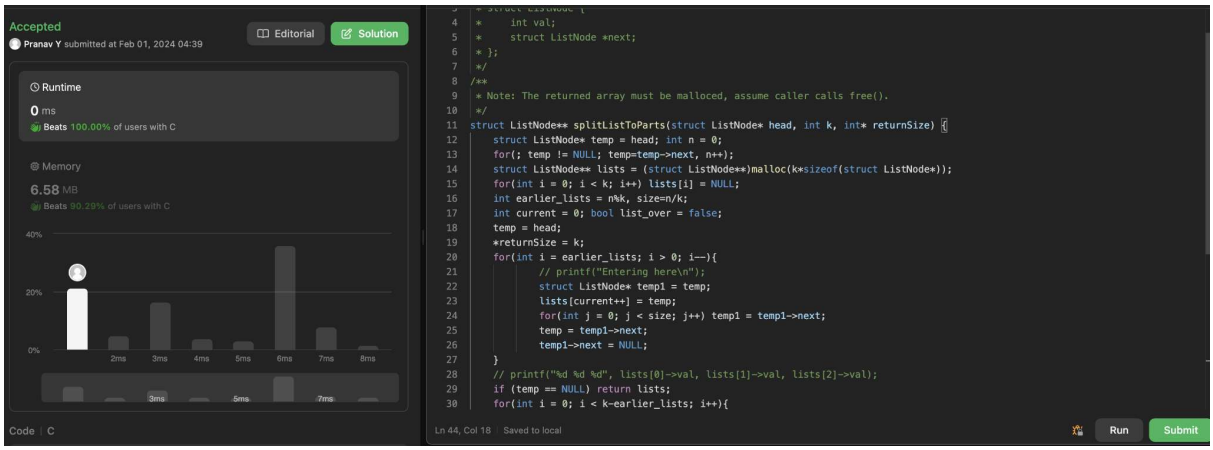
**Return an array of the k parts.**

**Code:**

```
struct ListNode** splitListToParts(struct ListNode* head, int k,
```

```
int* returnSize) {
    struct ListNode* temp = head; int n = 0;
    for(; temp != NULL; temp=temp->next, n++);
    struct ListNode** lists = (struct
ListNode**)malloc(k*sizeof(struct ListNode*));
    for(int i = 0; i < k; i++) lists[i] = NULL;
    int earlier_lists = n%k, size=n/k;
    int current = 0; bool list_over = false;
    temp = head;
    *returnSize = k;
    for(int i = earlier_lists; i > 0; i--){
            // printf("Entering here\n");
            struct ListNode* temp1 = temp;
            lists[current++] = temp;
            for(int j = 0; j < size; j++) temp1 = temp1->next;
            temp = temp1->next;
            temp1->next = NULL;
    }
    // printf("%d %d %d", lists[0]->val, lists[1]->val,
lists[2]->val);
    if (temp == NULL) return lists;
    for(int i = 0; i < k-earlier_lists; i++){
            struct ListNode* temp1 = temp;
            if (temp1 == NULL) break;
            for(int j = 0; j < size-1; j++) temp1 = temp1->next;
            lists[current++] = temp;
            temp = temp1->next;
            temp1->next = NULL;
            // for(int l = 0; l < k; l++){
            //     for(struct ListNode* temp2 = lists[l]; temp2 !=
NULL; temp2 = temp2->next){
            //         printf("%d ", temp2->val);
            //     }
            //     printf("\n");
            // }
    }
    return lists;
}
```

**Output:**




**Lab 8**
**Write a program**

**a) To construct a binary Search tree.**
**b) To traverse the tree using all the methods i.e., in-order, preorder and post order**
**c) To display the elements in the tree.**

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Node{
    int data;
    struct Node *left;
    struct Node *right;
} node;

node *root = NULL;

void insert(node **root, int data);
void preorder(node **root);
void postorder(node **root);
void inorder(node **root);

int main(){
    int choice, data;
    insert(&root, 100);
    insert(&root, 20);
    insert(&root, 200);
    insert(&root, 10);
```

```
    insert(&root, 30);
    insert(&root, 150);
    insert(&root, 300);
    printf("1. Preorder\n2. Inorder\n3. Postorder\n4.
Exit\nChoice: ");
    scanf("%d", &choice);
    while (choice != 4){
        if (choice == 1){
            preorder(&root);
            printf("\n");
        } else if (choice == 2){
            inorder(&root);
            printf("\n");
        } else if (choice == 3){
            postorder(&root);
            printf("\n");
        }
        printf("Enter choice: ");
        scanf("%d", &choice);
    }
}

void insert(node **root, int data){
    if (*root == NULL) {
        node *new_node = malloc(sizeof(node));
        new_node->data = data;
        new_node->right = NULL;
        new_node->left = NULL;
        *root = new_node;
        return;
    }
    if (data < (*root)->data){
        insert(&((*root)->left), data);
    } else if (data > (*root)->data){
        insert(&((*root)->right), data);
    }
    return;
}

void preorder(node **root){
    if (*root != NULL){
        printf("%d ", (*root)->data);
```

```
        preorder(&((*root)->left));
        preorder(&((*root)->right));
    }
}

void postorder(node **root){
    if (*root != NULL){
        postorder(&((*root)->left));
        postorder(&((*root)->right));
        printf("%d ", (*root)->data);
    }
}

void inorder(node **root){
    if (*root != NULL) {
        inorder(&(*root)->left);
        printf("%d ", (*root)->data);
        inorder(&(*root)->right);
    }
}
```

**Output:**

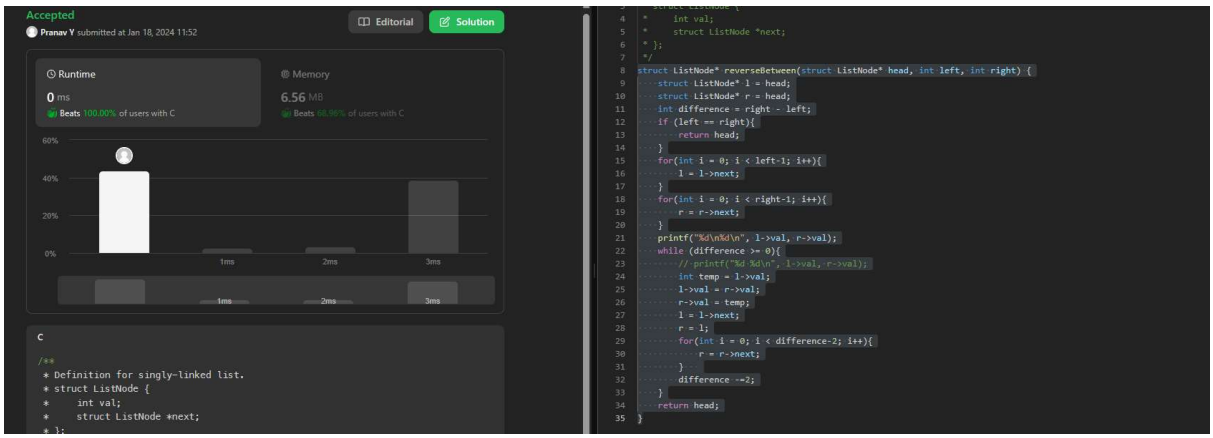
```
1. Preorder
2. Inorder
3. Postorder
4. Exit
Choice: 1
100 20 10 30 200 150 300
Enter choice: 2
10 20 30 100 150 200 300
Enter choice: 3
10 30 20 150 300 200 100
Enter choice: 4

Process returned 0 (0x0)   execution time : 7.360 s
Press any key to continue.
```

**Leetcode**
**Given the head of a linked list, rotate the list to the right by k places.**

**Code:**

```
struct ListNode* rotateRight(struct ListNode* head, int k) {
    struct ListNode *temp = head;
    if (head == NULL) return NULL;
    if (head->next == NULL) return head;
    if (k == 0) return head;
    int size = 1;
    for(; temp->next != NULL; temp=temp->next, size++);
    k %= size;
    if (k == 0) return head;
    temp->next = head;
    struct ListNode *temp1 = head;
    for(int i = 0; i < (size-k-1); temp1 = temp1->next, i++);
    head = temp1->next;
    temp1->next = NULL;
    return head;
}
```

**Output:**



**Lab 9**

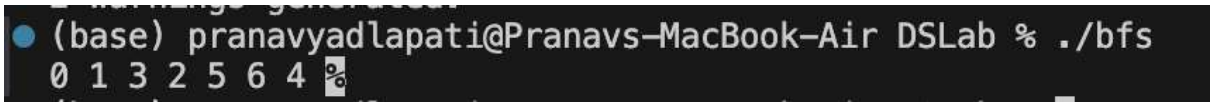
**a) Write a program to traverse a graph using BFS method.**

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define size 7

void push(int a);
int pop();
void display();
void bfs(int graph[][7]);

int fpos = -1, rpos = -1;
int queue[size];

int main(){
   int adj_matrix[7][7] = {
        {0, 1, 0, 1, 0, 0, 0},
        {1, 0, 1, 1, 0, 1, 1},
        {0, 1, 0, 1, 1, 1, 0},
        {1, 1, 1, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 1},
        {0, 1, 1, 0, 0, 0, 0},
        {0, 1, 0, 0, 1, 0, 0},
    };
    for(int i = 0; i < 7; i++) queue[i] = NULL;
    // display();
    bfs(adj_matrix);
    return 0;

}

void bfs(int graph[][7]){
    int visited[7];
    for(int i = 0; i < 7; i++) visited[i] = 0;
    push(0); visited[0]= 1;
    while (fpos != size){
        for(int i = 0; i < 7; i++){
            if(graph[queue[fpos]][i] == 1 && visited[i] != 1){
                push(i);
```

```
                visited[i] = 1;
                // break;
            }
        }
        printf("%d ", pop());
        // printf("%d\n", new_node);
    }
}

void push(int a){
    if (fpos == -1 && rpos == -1){
        queue[++rpos] = a;
        fpos++;
        return;
    }
    else if (rpos == size-1){
        printf("Queue overflow condition\n");
        return;
    }
    else{
        queue[++rpos] = a;
        return;
    }
}

int pop(){
    if (fpos == -1){
        printf("Queue Underflow condition\n");
    }
    int n = queue[fpos];
    queue[fpos] = (int) NULL;
    fpos++;
    return n;
}

void display(){
    printf("Queue: ");
    for(int i = 0; i < size; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
```

**Output:**



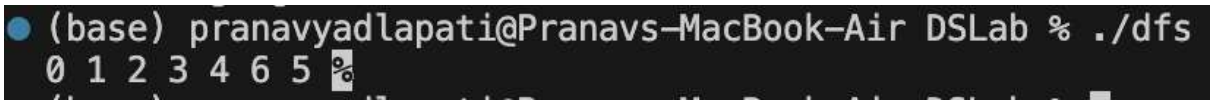

**9b) Write a program to check whether given graph is connected or not using DFS method.**

**Code:**

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

#define size 7
int pos = -1;
int stack[size];

void push(int a);
int pop();
void display();
void dfs(int graph[][7]);

int main(){
    int adj_matrix[7][7] = {
        {0, 1, 0, 1, 0, 0, 0},
        {1, 0, 1, 1, 0, 1, 1},
        {0, 1, 0, 1, 1, 1, 0},
        {1, 1, 1, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 1},
        {0, 1, 1, 0, 0, 0, 0},
        {0, 1, 0, 0, 1, 0, 0},
    };
    for(int i = 0; i < 7; i++) stack[i] = NULL;
    // display();
    dfs(adj_matrix);
    return 0;


}

void dfs(int graph[][7]){
    int visited[7];
    for (int i = 0; i < 7; i++) visited[i] = 0;
    push(0); visited[0] = 1; printf("0 ");
```

```
    while(pos != -1){
        bool new_node = false;
        for(int i = 0; i < 7; i++){

            if(graph[stack[pos]][i] == 1 && visited[i] != 1){
                new_node = true;
                push(i);

                visited[i] = 1; printf("%d ", i);
                break;
            }
        }

        if (!new_node) pop();
    }

}

void push(int a){
    if (pos == size-1){
        printf("Stack Overflow condition");
        return;
    }
    stack[++pos] = a;
}

int pop(){
    if (pos == -1){
        printf("Stack Underflow condition");
        return (int) NULL;
    }
    return stack[pos--];
}

void display(){
    for(int i = 0; i < size; i++){
        printf("%d ", stack[i]);
    }
    printf("\n");
}
```

**Output:**




**Hackerrank Problem:**

**Given a tree and an integer, k, in one operation, we need to swap the subtrees of all the nodes at each depth h, where h ∈ [k, 2k, 3k,...]. In other words, if h is a multiple of k, swap the left and right subtrees of that level.**

**You are given a tree of n nodes where nodes are indexed from [1..n] and it is rooted at 1. You have to perform t swap operations on it, and after each swap operation print the in-order traversal of the current state of the tree.**

**Code:**

```
struct node {
    int data;
    struct node *left;
    struct node *right;
};

struct node* create_node(int val){
    if(val == -1){
        return NULL;
    }
    struct node *temp=(struct node*)malloc(sizeof(struct node));
    temp->data=val;
    temp->left=NULL;
    temp->right=NULL;
    return temp;
}

void inorder(struct node *root){
    if(!root){
        return;
    }
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
```

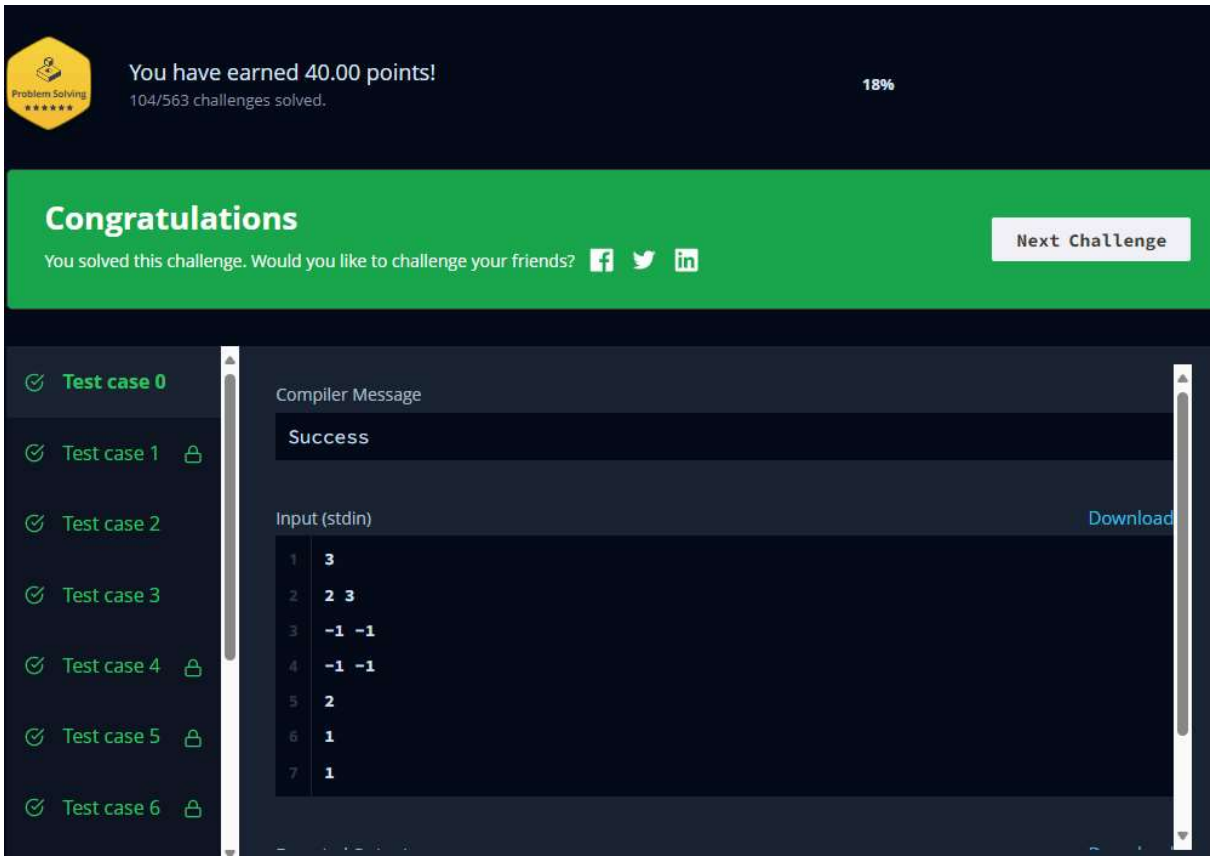
```
int max(int a, int b){
    if(a>b){
        return a;
    } else {
        return b;
    }
}

int height(struct node * root){
    if(!root){
        return 0;
    }
    return(1+max(height(root->left),height(root->right)));
}

void swap_nodes_at_level(struct node *root, int inc, int level,
int height){
    struct node *tnode;
    if(!root){
        return;
    }
    if(level > height){
        return;
    }
    if(!(level%inc)){
        tnode=root->left;
        root->left=root->right;
        root->right=tnode;
    }
    swap_nodes_at_level(root->left, inc, level+1, height);
    swap_nodes_at_level(root->right, inc, level+1, height);
}

int tail=0;
int head=0;


void enqueue(struct node **queue, struct node *root){

    queue[tail]=root;
    tail++;
}
```

```
struct node* dequeue(struct node **queue){

    struct node *temp = queue[head];
    head++;
    return temp;
}

int main() {

    /* Enter your code here. Read input from STDIN. Print output
to STDOUT */
    int nodes_count, i, temp, h, tc_num, index, inc, temp1, temp2;

    scanf("%d", &nodes_count);

  //  printf("%d\n", nodes_count);

    // int arr[2*nodes_count+1];

    struct node *root_perm, *root_temp;

    //queue=create_queue(nodes_count);

    struct node *q[nodes_count];
    for(i=0;i<nodes_count;i++){
        q[i]=NULL;
    }

    //building the array

    i=0,index=1;
    root_temp=root_perm=create_node(1);
    enqueue(q, root_temp);

    while(index<=2*nodes_count) {

        //printf("\n In Loop : i : %d",i);
        root_temp=dequeue(q);
        //setting up the left child
        scanf("%d", &temp1);
        if(temp1 == -1){
```

```
    } else {
        root_temp->left=create_node(temp1);
        enqueue(q, root_temp->left);
    }
    //setting up the right child
    scanf("%d", &temp2);
    if(temp2==-1) {

    } else {
        root_temp->right=create_node(temp2);
        enqueue(q, root_temp->right);
    }
    index=index+2;
  //  i++;

}

h = height(root_perm);

scanf("%d", &tc_num);

//printf("%d",tc_num);
//printf("\n");

//inorder(root_perm);

while(tc_num){
    scanf("%d",&inc);
    temp=inc;
    //while(temp < height){
    swap_nodes_at_level(root_perm, inc, 1, h);
    //temp=temp + inc;
    //}
    //temp=0;
    inorder(root_perm);
    printf("\n");
    tc_num--;
}

//Tree is created at this point
```

```
    return 0;
}
```

**Output:**



**Lab 10:**
**Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table (HT) of memory locations with L as the set of memory addresses (2-digit) of locations in HT.**
**Let the keys in K and addresses in L are integers.**

**Design and develop a Program in C that uses Hash function H: K> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L.**
**Resolve the collision (if any) using linear probing.**

**Code:**

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

#define size 10

int table[size];

void push(int data);
int pop(int data);
void search(int data);
void display();

int main(){
    for(int i = 0; i < size; i++) table[i] = -1;
    int choice;
    printf("1. Insert\n2. Search\n3. Delete\n4. Exit\nChoice: ");
    scanf("%d", &choice);
    int a;
    while(choice != 4){
        switch(choice){
            case 1:
                printf("Enter integer to be pushed: ");
                scanf("%d", &a);
                push(a);
                break;
            case 2:
                printf("Enter integer to be popped: ");
                scanf("%d", &a);
                int res = pop(a);
                if (res == 0) printf("Integer popped\n");
```

```
                else printf("Integer not found\n");
                break;
            case 3:
                display();
                break;
            default:
                printf("Idk");
                break;
        }
        printf("Enter choice: ");
        scanf("%d", &choice);
    }
}

void push(int data){
    int hash = data%size;
    while (table[hash] != -1 && hash <= (hash+size-1)) hash =
(hash+1)%size;
    if (table[hash] == -1) table[hash] = data;
    else printf("Table is full");
}

int pop(int data){
    int hash = data%size;
     for(int i = 0; (table[hash] != data) || (i < size); i++, hash
= (hash+1)%size);
     if (table[hash] == data) {
        table[hash] = -1; return 0;
     }
     return -1;
}

void display(){
    printf("Table: ");
    for(int i = 0; i < size; i++)
        printf("%d ", table[i]);
    printf("\n");
}
```

**Output:**

```
1. Insert
2. Search
3. Delete
4. Exit
Choice: 1
Enter integer to be pushed: 1
Enter choice: 1
Enter integer to be pushed: 3
Enter choice: 1
Enter integer to be pushed: 90
Enter choice: 3
Table: 90 1 -1 3 -1 -1 -1 -1 -1 -1
Enter choice: 1
Enter integer to be pushed: 13
Enter choice: 3
Table: 90 1 -1 3 13 -1 -1 -1 -1 -1
Enter choice: 1
Enter integer to be pushed: 4
Enter choice: 3
Table: 90 1 -1 3 13 4 -1 -1 -1 -1
Enter choice: 4

Process returned 0 (0x0)   execution time : 45.082 s
Press any key to continue.
```