# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**On**

**DATA STRUCTURES (23CS3PCDST)**

**Submitted by**

**SIDHVIN VIDYADHAR BURLI**
**(1BM21CS211)**

**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**
**Dec 2023- March 2024**

**B. M. S. College of Engineering,**
**Bull Temple Road, Bangalore 560019**
**(Affiliated To Visvesvaraya Technological University, Belgaum)**
**Department of Computer Science and Engineering**

This is to certify that the Lab work entitled **"DATA STRUCTURES"** carried out by **Sidhvin Vidyadhar Burli (1BM21CS211)**, 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 year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Data structures Lab - **(23CS3PCDST )**work prescribed for the said degree.

**Prof. Lakshmi Neelima**
Assistant Professor
Department of CSE
BMSCE, Bengaluru

**Dr. Jyothi S Nayak**
Professor and Head
Department of CSE
BMSCE, Bengaluru

## 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.**

**Input Code:**

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

#define MAX_SIZE 5

int stack[MAX_SIZE];
int top = -1;

void push(int value) {
    if (top == MAX_SIZE - 1) {
        printf("Stack overflow!\n");
        return;
    }
    stack[++top] = value;
    printf("Pushed %d into the stack.\n", value);
}

int pop() {
    if (top == -1) {
        printf("Stack is empty/underflow!\n");
        return -1;
    }
    int item = stack[top--];
    printf("The deleted element is %d.\n", item);
    return item;
}

void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack contents:\n");
    for (int i = top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}

int main() {
    int choice, item;

    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
```

```
            scanf("%d", &item);
            push(item);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

**Output**

```
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 5
Pushed 5 into the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 4
Pushed 4 into the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack contents:
4
5
```

```
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
The deleted element is 4.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack contents:
5

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting program.
```

**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)**

**Input Code :**

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX_SIZE 100

void push(char stack[], int *top, char value);

char pop(char stack[], int *top);

char peek(char stack[], int top);

int isEmpty(int top);

int precedence(char operator);

void infixToPostfix(char infix[], char postfix[]);

int main() {

   char infix[MAX_SIZE], postfix[MAX_SIZE];

   printf("Enter the infix expression: ");

   fgets(infix, MAX_SIZE, stdin);

   infix[strlen(infix) - 1] = '\0';

   infixToPostfix(infix, postfix);

   printf("Postfix expression: %s\n", postfix);

   return 0;

}


void push(char stack[], int *top, char value) {

   if (*top == MAX_SIZE - 1) {
```

```
        printf("Stack overflow!\n");
        exit(EXIT_FAILURE);
    }
    stack[++(*top)] = value;
}
char pop(char stack[], int *top) {
    if (isEmpty(*top)) {
        printf("Stack underflow!\n");
        exit(EXIT_FAILURE);
    }
    return stack[(*top)--];
}

char peek(char stack[], int top) {
    if (isEmpty(top)) {
        printf("Stack is empty!\n");
        exit(EXIT_FAILURE);
    }
    return stack[top];
}
int isEmpty(int top) {
    return top == -1;
}
int precedence(char operator) {
    switch (operator) {
        case '+':
        case '-':
```

```
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}
void infixToPostfix(char infix[], char postfix[]) {
    char stack[MAX_SIZE];
    int top = -1;
    int i = 0, j = 0;

    while (infix[i] != '\0') {
        if (infix[i] == '(') {
            push(stack, &top, infix[i]);
        } else if ((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z')) {
            postfix[j++] = infix[i];
        } else if (infix[i] == ')') {
            while (!isEmpty(top) && peek(stack, top) != '(') {
                postfix[j++] = pop(stack, &top);
            }
            if (!isEmpty(top) && peek(stack, top) != '(') {
                printf("Invalid infix expression!\n");
                exit(EXIT_FAILURE);
            } else {
                pop(stack, &top);
```

```
        }
      } else {
        while (!isEmpty(top) && precedence(infix[i]) <= precedence(peek(stack, top))) {
          postfix[j++] = pop(stack, &top);
        }
        push(stack, &top, infix[i]);
      }
      i++;
    }

    while (!isEmpty(top)) {
      postfix[j++] = pop(stack, &top);
    }
    postfix[j] = '\0';
}
```

**<u>Output</u>**

```
Enter the infix expression: (A+B)*C-(D/E)
Postfix expression: AB+C*DE/-
```

**Lab Program 3:**

**write a program to simulate the working of the queue of integers using an array. Provide the following operations: Insert, delete, display. The program should print appropriate messages for overflow and underflow condition.**

**Input Code:**

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 10


void insert(int queue[], int *rear, int value);

int delete(int queue[], int *front, int rear);

void display(int queue[], int front, int rear);

int isFull(int rear);

int isEmpty(int front, int rear);


int main() {

   int queue[MAX_SIZE];

   int front = -1, rear = -1;

   int choice, value;


   while (1) {

      printf("\nQueue Operations:\n");

      printf("1. Insert\n");

      printf("2. Delete\n");

      printf("3. Display\n");

      printf("4. Exit\n");

      printf("Enter your choice: ");
```

```
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(queue, &rear, value);
                break;
            case 2:
                value = delete(queue, &front, rear);
                if (value != -1)
                    printf("Deleted element: %d\n", value);
                break;
            case 3:
                display(queue, front, rear);
                break;
            case 4:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

```
void insert(int queue[], int *rear, int value) {
    if (isFull(*rear)) {
        printf("Queue overflow!\n");
        return;
    }
    if (*rear == -1) {
        queue[++(*rear)] = value;
    } else {
        queue[++(*rear)] = value;
    }
    printf("Inserted %d into the queue.\n", value);
}

int delete(int queue[], int *front, int rear) {
    if (isEmpty(*front, rear)) {
        printf("Queue underflow!\n");
        return -1;
    }
    return queue[(*front)++];
}

void display(int queue[], int front, int rear) {
    if (isEmpty(front, rear)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue contents:\n");
```

```
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}
int isFull(int rear) {
    return rear == MAX_SIZE - 1;
}
int isEmpty(int front, int rear) {
    return front > rear;
}
```

**Output**

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 15
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20
```

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
15 20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

**Lab Program 4:**

**write a program to simulate the working of a circular queue using an array. Provide the following operations: insert, delete& display. The program should print appropriate message for queue empty and queue overflow conditions.**

**Input Code:**

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

#define MAX_SIZE 5

int queue[MAX_SIZE];
int front = -1, rear = -1;

void enqueue(int value);
int dequeue();
void display();
int isFull();
int isEmpty();

int main() {
    int choice, value;

    while (1) {
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                value = dequeue();
                if (value != -1)
                    printf("Dequeued element: %d\n", value);
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting program.\n");
```

```
            exit(0);
          default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

void enqueue(int value) {
    if (isFull()) {
        printf("Queue overflow!\n");
        return;
    }
    if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % MAX_SIZE;
    }
    queue[rear] = value;
    printf("Enqueued %d into the queue.\n", value);
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow!\n");
        return -1;
    }
    int value = queue[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % MAX_SIZE;
    }
    return value;
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue contents:\n");
    int i = front;
    do {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (rear + 1) % MAX_SIZE);
    printf("\n");
}
```

```
int isFull() {
    return (rear + 1) % MAX_SIZE == front;
}

int isEmpty() {
    return front == -1;
}
```

**Output**

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 5
Enqueued 5 into the queue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued element: 5

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
Exiting program.
```

**Lab Program 5:**

**1.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.**

**Input Code:**

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

struct Node {
```

```
    int data;
    struct Node* next;
};
struct Node* createNode(int value);
struct Node* insertFirst(struct Node* head, int value);
struct Node* insertAtPosition(struct Node* head, int value, int position);
struct Node* insertEnd(struct Node* head, int value);
void display(struct Node* head);

int main() {
    struct Node* head = NULL;
    int choice, value, position;

    while (1) {
        printf("\nLinked List Operations:\n");
        printf("1. Insert at First\n");
        printf("2. Insert at Position\n");
        printf("3. Insert at End\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert at first: ");
                scanf("%d", &value);
                head = insertFirst(head, value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert: ");
                scanf("%d", &position);
                head = insertAtPosition(head, value, position);
                break;
            case 3:
                printf("Enter value to insert at end: ");
                scanf("%d", &value);
                head = insertEnd(head, value);
                break;
            case 4:
                display(head);
                break;
            case 5:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
```

```
    }

    return 0;
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

struct Node* insertFirst(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}

struct Node* insertAtPosition(struct Node* head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return head;
    }
    if (position == 1) {
        return insertFirst(head, value);
    }
    struct Node* newNode = createNode(value);
    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return head;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    return head;
}
struct Node* insertEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        return newNode;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
```

```
        temp = temp->next;
    }
    temp->next = newNode;
    return head;
}

void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked List: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

**Output**:

```
Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Display
5. Exit
Enter your choice: 1
Enter value to insert at first: 5

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Display
5. Exit
Enter your choice: 2
Enter value to insert: 6
Enter position to insert: 2

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Display
5. Exit
Enter your choice: 4
Linked List: 5 6
```

**Lab Program 6**:

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

**Input Code:**

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

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

struct Node* createNode(int value);
struct Node* insertFirst(struct Node* head, int value);
struct Node* insertAtPosition(struct Node* head, int value, int position);
struct Node* insertEnd(struct Node* head, int value);
struct Node* deleteFirst(struct Node* head);
struct Node* deleteSpecified(struct Node* head, int key);
struct Node* deleteLast(struct Node* head);
void display(struct Node* head);

int main() {
    struct Node* head = NULL;
    int choice, value, key, position;

    while (1) {
        printf("\nLinked List Operations:\n");
        printf("1. Insert at First\n");
        printf("2. Insert at Position\n");
        printf("3. Insert at End\n");
        printf("4. Delete First Element\n");
        printf("5. Delete Specified Element\n");
        printf("6. Delete Last Element\n");
        printf("7. Display\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert at first: ");
                scanf("%d", &value);
                head = insertFirst(head, value);
                break;
            case 2:
```

```
            printf("Enter value to insert: ");
            scanf("%d", &value);
            printf("Enter position to insert: ");
            scanf("%d", &position);
            head = insertAtPosition(head, value, position);
            break;
        case 3:
            printf("Enter value to insert at end: ");
            scanf("%d", &value);
            head = insertEnd(head, value);
            break;
        case 4:
            head = deleteFirst(head);
            break;
        case 5:
            printf("Enter the value to delete: ");
            scanf("%d", &key);
            head = deleteSpecified(head, key);
            break;
        case 6:
            head = deleteLast(head);
            break;
        case 7:
            display(head);
            break;
        case 8:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

struct Node* insertFirst(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
```

```
    return newNode;
}

struct Node* insertAtPosition(struct Node* head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return head;
    }
    if (position == 1) {
        return insertFirst(head, value);
    }
    struct Node* newNode = createNode(value);
    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return head;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    return head;
}

struct Node* insertEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        return newNode;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    return head;
}

struct Node* deleteFirst(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return NULL;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
    printf("Deleted first element.\n");
    return head;
}
```

```
struct Node* deleteSpecified(struct Node* head, int key) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return NULL;
    }
    if (head->data == key) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        printf("Deleted specified element.\n");
        return head;
    }
    struct Node* prev = NULL;
    struct Node* current = head;
    while (current != NULL && current->data != key) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Specified element not found.\n");
        return head;
    }
    prev->next = current->next;
    free(current);
    printf("Deleted specified element.\n");
    return head;
}

struct Node* deleteLast(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return NULL;
    }
    if (head->next == NULL) {
        free(head);
        printf("Deleted last element.\n");
        return NULL;
    }
    struct Node* prev = NULL;
    struct Node* current = head;
    while (current->next != NULL) {
        prev = current;
        current = current->next;
    }
    prev->next = NULL;
    free(current);
    printf("Deleted last element.\n");
    return head;
}
```

```
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked List: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

**Output:**

```
Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Delete First Element
5. Delete Specified Element
6. Delete Last Element
7. Display
8. Exit
Enter your choice: 7
Linked List: 8 7 6 5

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Delete First Element
5. Delete Specified Element
6. Delete Last Element
7. Display
8. Exit
Enter your choice: 4
Deleted first element.

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Delete First Element
5. Delete Specified Element
6. Delete Last Element
7. Display
```

```
7. Display
8. Exit
Enter your choice: 5
Enter the value to delete: 6
Deleted specified element.

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Delete First Element
5. Delete Specified Element
6. Delete Last Element
7. Display
8. Exit
Enter your choice: 7
Linked List: 7 5

Linked List Operations:
1. Insert at First
2. Insert at Position
3. Insert at End
4. Delete First Element
5. Delete Specified Element
6. Delete Last Element
7. Display
8. Exit
Enter your choice: 8
Exiting program.
```

**Lab Program 7:**

**Linked List -sort , reverse , concatenation.**

**Input Code:**

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

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Node* insertAtBeginning(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = head;
    return newNode;
}

void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

struct Node* sortLinkedList(struct Node* head) {
    struct Node *current, *index;
    int temp;
    if (head == NULL)
        return head;

    current = head;
    while (current != NULL) {
        index = current->next;
        while (index != NULL) {
```

```
            if (current->data > index->data) {
                temp = current->data;
                current->data = index->data;
                index->data = temp;
            }
            index = index->next;
        }
        current = current->next;
    }
    return head;
}

struct Node* reverseLinkedList(struct Node* head) {
    struct Node *prev = NULL, *current = head, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

struct Node* concatenateLinkedLists(struct Node* head1, struct Node* head2) {
    if (head1 == NULL)
        return head2;
    struct Node* temp = head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head2;
    return head1;
}

int main() {
    struct Node* head1 = NULL;
    head1 = insertAtBeginning(head1, 5);
    head1 = insertAtBeginning(head1, 3);
    head1 = insertAtBeginning(head1, 8);
    head1 = insertAtBeginning(head1, 1);
    printf("Original List 1: ");
    printList(head1);

    head1 = sortLinkedList(head1);
    printf("Sorted List 1: ");
    printList(head1);

    head1 = reverseLinkedList(head1);
    printf("Reversed List 1: ");
    printList(head1);
```

```
    struct Node* head2 = NULL;
    head2 = insertAtBeginning(head2, 9);
    head2 = insertAtBeginning(head2, 6);
    head2 = insertAtBeginning(head2, 2);
    printf("Original List 2: ");
    printList(head2);

    head2 = sortLinkedList(head2);
    printf("Sorted List 2: ");
    printList(head2);

    head2 = reverseLinkedList(head2);
    printf("Reversed List 2: ");
    printList(head2);

    head1 = concatenateLinkedLists(head1, head2);
    printf("Concatenated List: ");
    printList(head1);

    return 0;
}
```

**Output**:

```
Original List 1: 1 8 3 5
Sorted List 1: 1 3 5 8
Reversed List 1: 8 5 3 1
Original List 2: 2 6 9
Sorted List 2: 2 6 9
Reversed List 2: 9 6 2
Concatenated List: 8 5 3 1 9 6 2
```

**Lab Program 8:**

a. **Stack implementation using single linked list.**
b. **Queue implementation using single linked list.**

**Input Code:**

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

```
struct node {
   int info;
   struct node *ptr;
}*top,*top1,*temp;

int count = 0;
// Push() operation on a  stack
void push(int data) {
    if (top == NULL)
   {
      top =(struct node *)malloc(1*sizeof(struct node));
      top->ptr = NULL;
      top->info = data;
   }
   else
   {
      temp =(struct node *)malloc(1*sizeof(struct node));
      temp->ptr = top;
      temp->info = data;
      top = temp;
   }
   count++;
   printf("Node is Inserted\n\n");
}



int pop() {
    top1 = top;

   if (top1 == NULL)
   {
      printf("\nStack Underflow\n");
      return -1;
   }
   else
      top1 = top1->ptr;
   int popped = top->info;
   free(top);
   top = top1;
   count--;
   return popped;
}



void display() {
   // Display the elements of the stack
```

```
    top1 = top;

    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
        return;
    }

    printf("The stack is \n");
    while (top1 != NULL)
    {
        printf("%d--->", top1->info);
        top1 = top1->ptr;
    }
    printf("NULL\n\n");

}




int main() {
    int choice, value;
    printf("\nImplementation of Stack using Linked List\n");
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("\nEnter the value to insert: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            printf("Popped element is :%d\n", pop());
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
            break;
        default:
            printf("\nWrong Choice\n");
        }
    }
}
```

**Output Code:**

```
Implementation of Stack using Linked List

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1

Enter the value to insert: 5
Node is Inserted


1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1

Enter the value to insert: 6
Node is Inserted
```

```
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 2
Popped element is :6

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 3
The stack is
5--->NULL
```

**Lab program 9:**

**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.**

**Input Code:**

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

struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
```

```
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void insertLeft(struct Node** headRef, struct Node* targetNode, int data) {
    struct Node* newNode = createNode(data);
    if (targetNode == NULL) {
        printf("Cannot insert left of a NULL node\n");
        return;
    }
    newNode->next = targetNode;
    newNode->prev = targetNode->prev;
    if (targetNode->prev != NULL) {
        targetNode->prev->next = newNode;
    } else {
        *headRef = newNode;
    }
    targetNode->prev = newNode;
}

void deleteNode(struct Node** headRef, int key) {
    struct Node* current = *headRef;
    struct Node* temp = NULL;

    while (current != NULL && current->data != key) {
        current = current->next;
    }

    if (current == NULL) {
        printf("Node with key %d not found\n", key);
        return;
    }

    if (current == *headRef) {
        *headRef = current->next;
    }

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

    free(current);
}

void printList(struct Node* head) {
```

```
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    head = createNode(6);
    head->next = createNode(7);
    head->next->prev = head;
    head->next->next = createNode(98);
    head->next->next->prev = head->next;

    printf("Doubly Linked List: ");
    printList(head);

    insertLeft(&head, head->next, 5);
    printf("After inserting 5 to the left of node with data 2: ");
    printList(head);

    deleteNode(&head, 98);
    printf("After deleting node with data 2: ");
    printList(head);

    return 0;
}
```

**Output :**

```
Doubly Linked List: 6 7 98
After inserting 5 to the left of node with data 2: 6 5 7 98
After deleting node with data 2: 6 5 7
```

### Leetcode Program 1:

```
int scoreOfParentheses(char* s) {
    int top=0, ans=0;
        for(int i=0; i<strlen(s); i++)
        {
          if(s[i]=='(') {
            top++;
```

```
            if(s[i+1]==')') {
                ans+=pow(2,top-1);
            }
          }
          else top-- ;
        }
        return ans;
    }
```


**<u>Lab program 10:</u>**

**Write a program.**

**a. To construct Binary Search tree**

**b. Traverse the tree using inorder , postorder, preorder.**

**c. Display the elements in the tree.**

**Input Code:**

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

struct Node {
    int data;
    struct Node *left;
```

```
    struct Node *right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
```

```
void displayTree(struct Node* root) {
    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\n");
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Elements in the tree:\n");
    displayTree(root);

    return 0;
}
```

**Output:**

```
Output
/tmp/NdIiBSHkkD.o
Elements in the tree:
Inorder traversal: 20 30 40 50 60 70 80
Postorder traversal: 20 40 30 60 80 70 50
Preorder traversal: 50 30 20 40 70 60 80
```

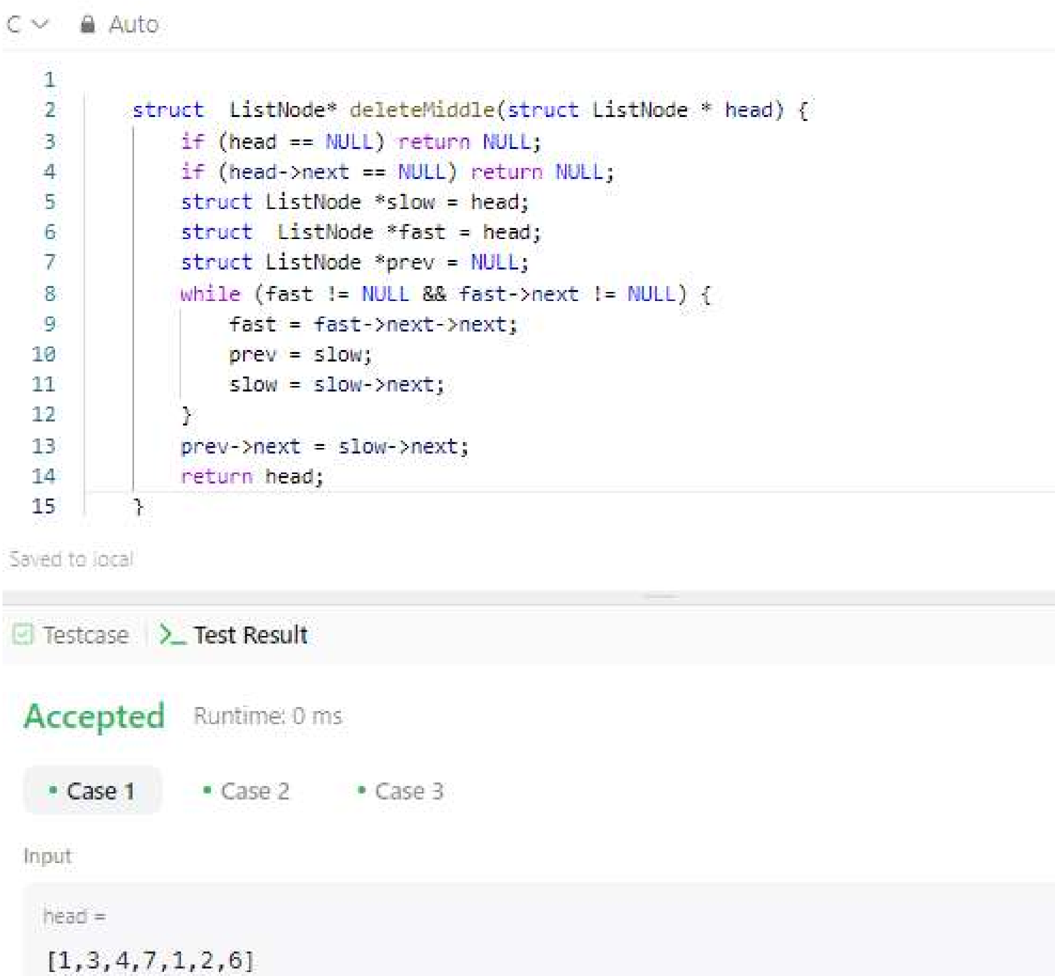
**LeetCode Program 2:**

**Delete the Middle Node of a Linked List**

**Input code:**

```
struct  ListNode* deleteMiddle(struct ListNode * head) {
    if (head == NULL) return NULL;
    if (head->next == NULL) return NULL;
    struct ListNode *slow = head;
    struct  ListNode *fast = head;
    struct ListNode *prev = NULL;
    while (fast != NULL && fast->next != NULL) {
        fast = fast->next->next;
        prev = slow;
        slow = slow->next;
    }
    prev->next = slow->next;
    return head;
}
```

**Output:**



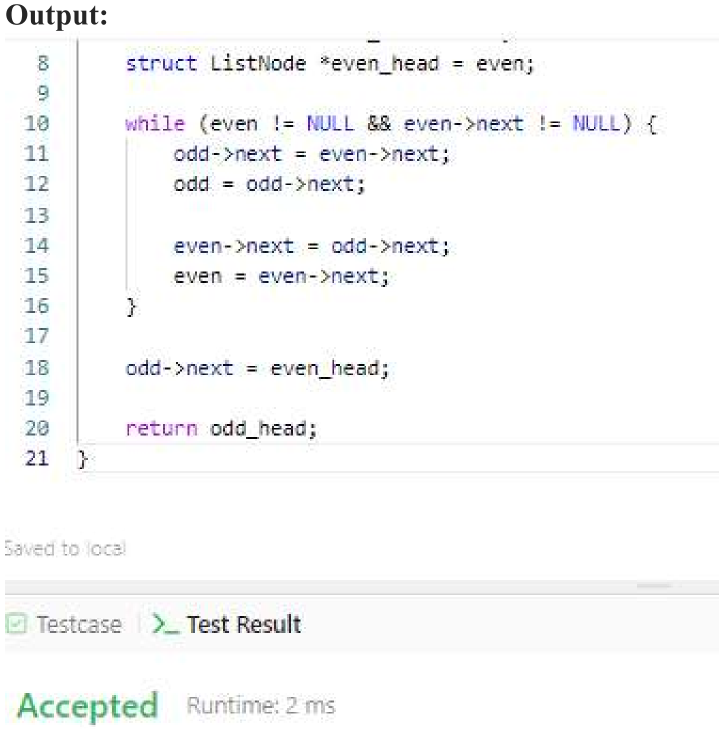
**LeetCode Program 3:**

**Odd Even Linked List- Leet code.**

**Input :**

```
struct ListNode* oddEvenList(struct ListNode* head) {
    if (head == NULL || head->next == NULL || head->next->next == NULL)
        return head:
    struct ListNode *odd = head;
    struct ListNode *even = head->next;
    struct ListNode *odd_head = odd;
    struct ListNode *even_head = even;

    while (even != NULL && even->next != NULL) {
        odd->next = even->next;
        odd = odd->next;

        even->next = odd->next;
        even = even->next;
    }

    odd->next = even_head;

    return odd_head;
}
```

**Output:**



**Lab program 11:**

**WAP for BFS and DFS**

**Input code:**

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

#define MAX_NODES 100

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

struct Graph {
    int numVertices;
    struct Node* adjLists[MAX_NODES];
    int visited[MAX_NODES];
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```
void BFS(struct Graph* graph, int startNode) {
    int queue[MAX_NODES];
    int front = 0, rear = 0;

    graph->visited[startNode] = 1;
    queue[rear++] = startNode;

    while (front < rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->data;
            if (!graph->visited[adjVertex]) {
                graph->visited[adjVertex] = 1;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

void DFS(struct Graph* graph, int startNode) {
    graph->visited[startNode] = 1;
    printf("%d ", startNode);

    struct Node* temp = graph->adjLists[startNode];
    while (temp) {
        int adjVertex = temp->data;
        if (!graph->visited[adjVertex]) {
            DFS(graph, adjVertex);
        }
        temp = temp->next;
    }
}

int main() {
    struct Graph* graph = createGraph(4);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);

    printf("BFS starting from vertex 0: ");
    BFS(graph, 0);
    printf("\n");

    for (int i = 0; i < graph->numVertices; i++) {
```

```
        graph->visited[i] = 0;
    }

    printf("DFS starting from vertex 0: ");
    DFS(graph, 0);
    printf("\n");

    return 0;
}
```

**Output:**

```
Output
/tmp/NdIiBSHkkD.o
BFS starting from vertex 0: 0 2 1 3
DFS starting from vertex 0: 0 2 3 1
```

**LeetCode program 4:**

**Delete a node in BST.**
**Input Code:**

```
struct TreeNode* smallest(struct TreeNode* root)
{
    struct TreeNode * cur=root;
    while(cur->left!=NULL)
    {
        cur=cur->left;
    }
    return cur;
}
struct TreeNode* deleteNode(struct TreeNode* root, int key)
{

    if(root==NULL)
    {
        return root;
    }
    if(key < root->val)
    {
        root->left = deleteNode(root->left,key);
    }
    else if(key > root->val)
    {
        root->right=deleteNode(root->right,key);
    }
```

```
    else
    {
        if (root->left == NULL)
        {
            struct TreeNode *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct TreeNode *temp = root->left;
            free(root);
            return temp;
        }
        struct TreeNode *temp = smallest(root->right);
        root->val = temp->val;
        root->right = deleteNode(root->right, root->val);
    }
    return root;
}
```

**Output :**

**LeetCode Program 5:**

**Bottom left tree value**

**Input Code:**

```
void f(struct TreeNode *curr, int lev, int *ans, int *maxLev) {
    if (curr == NULL) return;
    if (lev > *maxLev) {
        *maxLev = lev;
        *ans = curr->val;
    }
    f(curr->left, lev + 1, ans, maxLev);
    f(curr->right, lev + 1, ans, maxLev);
}

int findBottomLeftValue(struct TreeNode *root) {
    int ans = root->val;
    int maxLev = 0;
    f(root, 0, &ans, &maxLev);
    return ans;
}
```

**Output:**