

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT On

DATA STRUCTURES (23CS3PCDST)

Submitted by

JAYASHREE TARAI (1BM24CS407)

**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
September 2024-January 2025**

**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 JAYASHREE TARAI (**1BM24CS407**), 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 2024-25. 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.

Dr. Selva kumar S
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1	Implementation of Stack and its operations	
2	Implementation of Infix to Postfix	
3	Implementation of Linear Queue	
4	Implementation of Circular Queue	
5	Implementation of singly linked list (insertions,display)	
6	Implementation of Delete operations on singly linked list	
7	Singly linked list operations(Reverse,sort,concatenate,stack ,queue)	
8	Doubly linked list with primitive operation	
9	Implementation on BST,DFS operations	
10	Hashing	

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.

```
#include <stdio.h>
#include <conio.h>
#define size 5

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

void push(int value){
    if (top== size-1){
        printf("stack overflow stack is full ");
    }
    else{
        top++;
        stack[top]=value;
        printf("\n insertion successful%d",value);
    }
}

void pop(){
    if(top == -1){
        printf("stack underflow stack is empty ");
    }
    else{
        printf("%d deleted element is ",stack[top]);
        top--;
    }
}

void display(){
    if(top == -1){
        printf("stack underflow stack is empty ");
    }
    else{
        printf("the elements are ");
        for(int i= top; i>=0; i--)
            printf("%d\n ",stack[i]);
    }
}
```

```

void main(){
    int choice,value;

    while(1){
        printf("\n\n**menu**\n");
        printf("the available options are \n");
        printf("1.push\n2.pop\n3.display\n4.exit");
        printf("\n enter ur choice ");
        scanf("%d",&choice);
        switch (choice){
            case 1:
                printf("\nenter valur \n");
                scanf("%d",&value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("--exit--");
            default:
                printf("invalid option try again");
        }

    }
}

```

output:

Output:

C:\Users\STUDENT\Desktop\Jayashree407CS\LAB1.exe

```
**menu**  
the available options are  
1.push  
2.pop  
3.display  
4.exit  
enter ur choice 1
```

```
enter value:  
10
```

```
insertion successful10
```

```
**menu**  
the available options are  
1.push  
2.pop  
3.display  
4.exit  
enter ur choice 1
```

```
enter value:  
20
```

```
insertion successful20
```

```
**menu**  
the available options are  
1.push  
2.pop  
3.display  
4.exit  
enter ur choice 3  
the elements are 20  
10
```

```
**menu**  
the available options are  
1.push  
2.pop  
3.display  
4.exit
```

```
enter ur choice 2  
deleted element is 20
```

```
**menu**  
the available options are  
1.push  
2.pop  
3.display  
4.exit  
enter ur choice 3  
the elements are 10
```

Lab program 2

2a. Write a program 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 <ctype.h>
#include <string.h>

#define MAX 100
char stack[MAX];
int top = -1;

void push(char c) {
    if (top == MAX - 1) {
        printf("Stack overflow\n");
    } else {
        stack[++top] = c;
    }
}

char pop() {
    if (top == -1) {
        printf("Stack underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}

int precedence(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/') {
        return 2;
    }
}
```

```
    return 0;
}
```

```
void infixToPostfix(char* expression) {
    char postfix[MAX];
    int i, j = 0;

    for (i = 0; expression[i] != '\0'; i++) {
        char ch = expression[i];

        if (isalnum(ch)) {
            postfix[j++] = ch;
        }

        else if (ch == '(') {
            push(ch);
        }

        else if (ch == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = pop();
            }
            pop(); // Remove '(' from the stack
        }

        else {
            while (top != -1 && precedence(stack[top]) >= precedence(ch)) {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }

    while (top != -1) {
        postfix[j++] = pop();
    }
}
```



```

    postfix[j] = '\0';

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

int main() {
    char expression[MAX];

    printf("Enter a valid infix expression: ");
    scanf("%s", expression);

    infixToPostfix(expression);

    return 0;
}

```

Output:

```

Enter a valid infix expression: A+B*(C-D)

Postfix expression: ABCD-*+

=== Code Execution Successful ===

```

2b.Demonstration of account creation on LeetCode platform

Program - Leetcode platform

account creation is done

Lab program 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

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

```
#define MAX 5
```

```
int queue[MAX];
```

```
int front = -1, rear = -1;
```

```
void insert(int value) {
```

```
    if (rear == MAX - 1) {
```

```
        printf("Queue Overflow! Cannot insert %d\n", value);
```

```
    } else {
```

```
        if (front == -1) {
```

```
            front = 0; // Set front to 0 when the first element is inserted
```

```
        }
```

```
        rear++;
```

```
        queue[rear] = value;
```

```
        printf("%d inserted into the queue\n", value);
```

```
    }
```

```
}
```

```
void delete() {
```

```
    if (front == -1 || front > rear) {
```

```
        printf("Queue Underflow! Cannot delete from an empty queue\n");
```

```
    } else {
```

```
        printf("%d deleted from the queue\n", queue[front]);
```

```
        front++;
```

```

        if (front > rear) {

            front = rear = -1;
        }
    }
}

```

```

void display() {
    if (front == -1) {
        // Queue empty condition
        printf("The queue is empty\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

```

```

int main() {
    int choice, value;

```

```

    do {
        printf("\nQueue Operations:\n");
        printf("1. Insert (Enqueue)\n");
        printf("2. Delete (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 insert: ");
                scanf("%d", &value);
                insert(value); // Call insert function
                break;

```

```

        case 2:
            delete(); // Call delete function
            break;
        case 3:
            display(); // Call display function
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }
} while (choice != 4);

return 0;
}

```

Output:

```

C:\Users\STUDENT\Desktop\Jayashree407CS\lab3.exe
Queue Operations:
1. Insert (Enqueue)
2. Delete (Dequeue)
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 10
10 inserted into the queue

Queue Operations:
1. Insert (Enqueue)
2. Delete (Dequeue)
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 20
20 inserted into the queue

Queue Operations:
1. Insert (Enqueue)
2. Delete (Dequeue)
3. Display
4. Exit
Enter your choice: 3
Queue elements: 10 20

Queue Operations:
1. Insert (Enqueue)
2. Delete (Dequeue)
3. Display
4. Exit
Enter your choice: 2
10 deleted from the queue

```

```

Queue Operations:
1. Insert (Enqueue)
2. Delete (Dequeue)
3. Display
4. Exit
Enter your choice: 3
Queue elements: 20

```

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

```
#include <stdio.h>
#define MAX 3
int queue[MAX];
int front = -1, rear = -1;

void insert(int value) {
    if ((front == 0 && rear == MAX - 1) || (rear == (front - 1) % (MAX - 1))) {
        printf("Queue Overflow\n");
        return;
    } else if (front == -1) {
        front = rear = 0;
        queue[rear] = value;
    } else if (rear == MAX - 1 && front != 0) {
        rear = 0;
        queue[rear] = value;
    } else {
        rear++;
        queue[rear] = value;
    }
    printf("Inserted element: %d\n", value);
}

void delete() {
    if (front == -1) {
        printf("Queue Underflow\n");
        return;
    }
    printf("Deleted element: %d\n", queue[front]);
    if (front == rear) {
        front = rear = -1;
    } else if (front == MAX - 1) {
```

```

        front = 0;
    } else {
        front++;
    }
}

```

```

void display() {
    if (front == -1) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Queue elements are: ");
    if (rear >= front) {
        for (int i = front; i <= rear; i++)
            printf("%d ", queue[i]);
    } else { // Circular condition
        for (int i = front; i < MAX; i++)
            printf("%d ", queue[i]);
        for (int i = 0; i <= rear; i++)
            printf("%d ", queue[i]);
    }
    printf("\n");
}

```

```

int main() {
    int choice, value;

    while (1) {
        printf("\nCircular Queue 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:

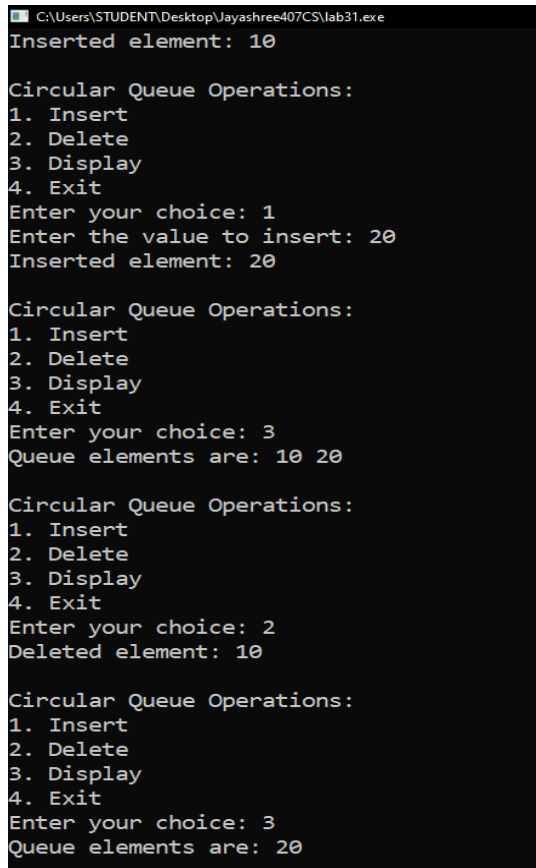
```

```

        printf("Enter the value to insert: ");
        scanf("%d", &value);
        insert(value);
        break;
    case 2:
        delete();
        break;
    case 3:
        display();
        break;
    case 4:
        printf("Exiting...\n");
        return 0;
    default:
        printf("Invalid choice! Please try again.\n");
    }
}
return 0;
}

```

Output:



```

C:\Users\STUDENT\Desktop\Jayashree407CS\lab31.exe
Inserted element: 10

Circular Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 20
Inserted element: 20

Circular Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements are: 10 20

Circular Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted element: 10

Circular Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements are: 20

```

Lab program 4

4a) 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.

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

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

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

```
void insertAtFront(struct Node** head, int newData) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = newData;  
    newNode->next = *head;  
    *head = newNode;  
}
```

```
void insertAtEnd(struct Node** head, int newData) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    struct Node* last = *head;  
  
    newNode->data = newData;  
    newNode->next = NULL;
```

```
    if (*head == NULL) {  
        *head = newNode;  
        return;  
    }
```

```
    while (last->next != NULL) {  
        last = last->next;  
    }
```

```
    last->next = newNode;  
}
```



```

void insertAtPosition(struct Node** head, int position, int newData) {
    if (position < 1) { // Invalid position (positions start from 1)
        printf("Invalid position\n");
        return;
    }

```

```

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head;

```

```

    newNode->data = newData;

```

```

    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

```

```

    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

```

```

    if (temp == NULL) {
        printf("Position exceeds the length of the list\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

```

```

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

```

```

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

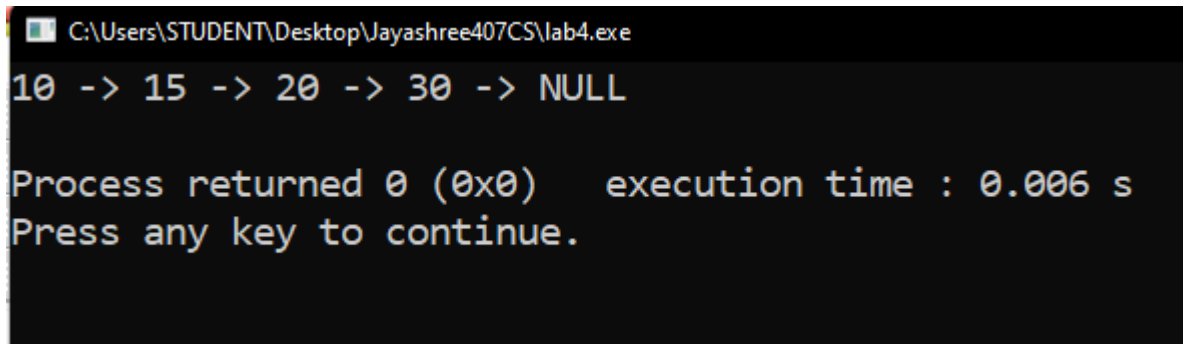
```

```
insertAtFront(&head, 10);
insertAtEnd(&head, 20);
insertAtEnd(&head, 30);
insertAtPosition(&head, 2, 15);

printList(head);

return 0;
}
```

output:



```
C:\Users\STUDENT\Desktop\Jayashree407CS\lab4.exe
10 -> 15 -> 20 -> 30 -> NULL

Process returned 0 (0x0)   execution time : 0.006 s
Press any key to continue.
```

4b) Program - Leetcode platform

Given two strings *s* and *t*, return true if they are equal when both are typed into empty text editors. '#' means a backspace character. Note that after backspacing an empty text, the text will continue empty.

Example 1: Input: *s* = "ab#c", *t* = "ad#c" Output: true Explanation: Both *s* and *t* become "ac".

Example 2: Input: *s* = "ab##", *t* = "c#d#" Output: true Explanation: Both *s* and *t* become "". **Example 3:** Input: *s* = "a#c", *t* = "b" Output: false Explanation: *s* becomes "c" while *t* becomes "b". **Constraints:** $1 \leq s.length, t.length \leq 200$ *s* and *t* only contain lowercase letters and '#' characters.

```
bool backspaceCompare(char* s, char* t) {
    int i = strlen(s) - 1, j = strlen(t) - 1;

    // Process both strings until we either reach the beginning of both strings
    while (i >= 0 || j >= 0) {
        // Process string s
        int skip_s = 0; // Track the number of backspaces to skip in s
        while (i >= 0 && (s[i] == '#' || skip_s > 0)) {
            if (s[i] == '#') {
                skip_s++; // We found a backspace, so increment skip count
            } else {
                skip_s--; // We skip this character
            }
            i--; // Move to the previous character
        }

        // Process string t
        int skip_t = 0; // Track the number of backspaces to skip in t
        while (j >= 0 && (t[j] == '#' || skip_t > 0)) {
            if (t[j] == '#') {
                skip_t++; // We found a backspace, so increment skip count
            } else {
                skip_t--; // We skip this character
            }
        }
    }
}
```

```

        j--; // Move to the previous character
    }

    // Compare the current characters in both strings
    if (i >= 0 && j >= 0 && s[i] != t[j]) {
        return false; // If characters don't match, return false
    }

    // If only one string has been completely processed, return false
    if ((i >= 0) != (j >= 0)) {
        return false; // One string is longer than the other after processing
    }

    i--; // Move to the previous character in string s
    j--; // Move to the previous character in string t
}

return true; // If we exit the loop without returning false, the strings are equal
}

```

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```

s =
"a#c"

```

4c) leetcode program

You are given a string number representing a positive integer and a character digit. Return the resulting string after removing exactly one occurrence of digit from number such that the value of the resulting string in decimal form is maximized.

The test cases are generated such that digit occurs at least once in number. Example 1:
Input: number = "123", digit = "3"

Output: "12" **Explanation:** There is only one '3' in "123"

. After removing '3', the result is "12".

```

char* removeDigit(char* number, char digit) {

    int len = strlen(number);

    static char result[101]; // Static array to store the result (to
    handle return value)

    int removeIndex = -1;

    // Iterate through the string to find the best position to remove the
    digit

    for (int i = 0; i < len - 1; i++) {

        if (number[i] == digit && number[i] < number[i + 1]) {

            // If the current digit is smaller than the next digit, remove
            this digit

            removeIndex = i;

            break;

        }

    }

    // If no such position found, remove the last occurrence of the digit

    if (removeIndex == -1) {

        for (int i = len - 1; i >= 0; i--) {

            if (number[i] == digit) {

                removeIndex = i;

                break;

            }

        }

    }

    // Construct the result string by skipping the character at removeIndex

```

```
int j = 0;

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

    if (i != removeIndex) {

        result[j++] = number[i];

    }

}

result[j] = '\0'; // Null-terminate the resulting string


return result;
}
```

Accepted Runtime: 0 ms

• Case 1 • **Case 2** • Case 3

Input

number =
"1231"

digit =
"1"

Output

"231"

Expected

"231"

Lab program 5

5a) 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.

```
#include <stdio.h>

#include <stdlib.h>

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

void insfrt(struct node** head,int ndata){
    struct node* newnode=(struct node*) malloc(sizeof(struct node));
    newnode->data=ndata;
    newnode->next=*head;
    *head=newnode;
}

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

void delfrt(struct node** head){
    struct node *ptr;
    if(head == NULL)
    {
```

```

printf("\nList is empty");
}
else
{
ptr = *head;
*head = ptr->next;
free(ptr);
printf("\n Node deleted from the begining ...");
}
}

void delend(struct node** head){
struct node *ptr,*prv;
if(head == NULL)
{
printf("\nlist is empty");
}
else if((*head)->next == NULL)
{
free(head);
head = NULL;
printf("\nOnly node of the list deleted ...");
}
else
{
struct node *ptr = *head;
while(ptr->next != NULL)
{
prv = ptr;
ptr = ptr ->next;

```



```

}

prv->next = NULL;

free(ptr);

printf("\n Deleted Node from the last ...");

}

}

void delpos(struct node** head){

struct node *ptr, *prv;

int loc,i;

printf("enter location :");

scanf("%d",&loc);

ptr=*head;

for(i=0;i<loc;i++)

{

prv = ptr;

ptr = ptr->next;

if(ptr == NULL)

{

printf("\nThere are less than %d elements in the list..\n",loc);

return;

}

}

prv ->next = ptr ->next;

free(ptr);

printf("\nDeleted %d node ",loc);

}

int main(){

struct node* head=NULL;

int choice,newdata;

```

```

while(1){

printf("the operation are 1.INSERT \n 2.Display \n 3.deltete at front \n 4.delete at end \n
5.delete at position\n");

printf("enter choice: ");

scanf("%d",&choice);

switch(choice){

case 1:

printf("Enter the number: ");

scanf("%d",&newdata);

insfrt(&head,newdata);

break;


case 2:

display(head);

break;

case 3:

delfrt(&head);

break;

case 4:

delend(&head);

break;

case 5:

delpos(&head);

break;

default:

printf("invalid choice");

}

}

return 0;}

```

output:

```
C:\Users\STUDENT\Desktop\Jayashree407CS\lab5.exe
3.deltete at front
4.delete at end
5.delete at position
enter choice: 1
Enter the number: 20
the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 1
Enter the number: 30
the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 2
30
20
10
the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 3

Node deleted from the begining ...the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 4

Deleted Node from the last ...the operation are 1.INSERT

Deleted Node from the last ...the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 5
enter location :0

Deleted 0 node the operation are 1.INSERT
2.Display
3.deltete at front
4.delete at end
5.delete at position
enter choice: 2
20
the operation are 1.INSERT
```

\

5b) Leetcode program

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Example 1: Input: head = [1,1,2] Output: [1,2]

Example 2: Input: head = [1,1,2,3,3] Output: [1,2,3]

Constraints: The number of nodes in the list is in the range [0, 300]. $-100 \leq \text{Node.val} \leq 100$ The list is guaranteed to be sorted in ascending order.

```
* Definition for singly-linked list.

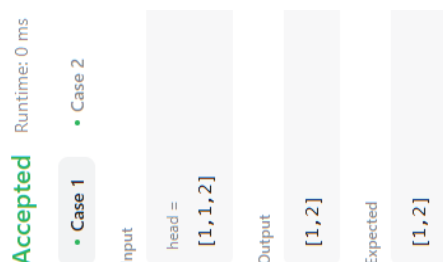
* struct ListNode {
*     int val;
*     struct ListNode *next;
* }

struct ListNode* deleteDuplicates(struct ListNode* head) {
    struct ListNode* current = head;

    while (current != NULL && current->next != NULL) {
        // If the current node's value is equal to the next node's value,
        skip the next node

        if (current->val == current->next->val) {
            current->next = current->next->next; // Remove the duplicate
        } else {
            current = current->next; // Move to the next node
        }
    }

    return head; // Return the modified list
}
```



Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter. Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1: Input: head = [3,2,0,-4], pos = 1 Output: true Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2: Input: head = [1,2], pos = 0 Output: true Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

```
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     struct ListNode *next;
* };

bool hasCycle(struct ListNode *head) {

    // Edge case: if the list is empty or contains only one node, there can't
    be a cycle

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

        return false;

    }

    // Initialize slow and fast pointers

    struct ListNode *slow = head;

    struct ListNode *fast = head;

    // Traverse the list with slow and fast pointers

    while (fast != NULL && fast->next != NULL) {

        slow = slow->next;           // Move slow pointer one step

        fast = fast->next->next;      // Move fast pointer two steps
```

```

        // If slow and fast pointers meet, there is a cycle

        if (slow == fast) {

            return true;

        }

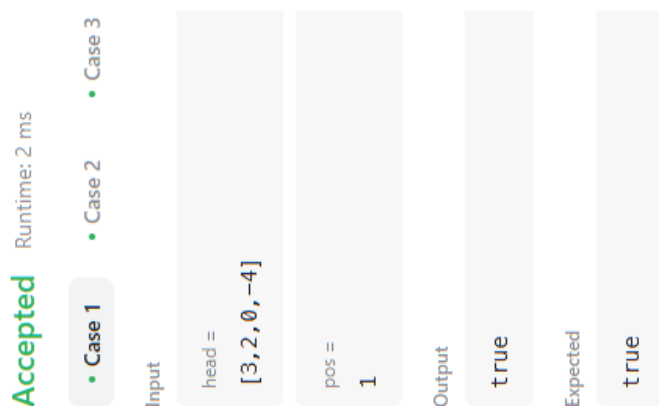
    }

    // If fast pointer reaches the end of the list, there is no cycle

    return false;

}

```



Given the head of a singly linked list, return true if it is a palindrome or false otherwise.
Example 1: Input: head = [1,2,2,1] Output: true **Example 2: Input: head = [1,2] Output: false**
Constraints: The number of nodes in the list is in the range [1, 105]. 0 <= Node.val <= 9

```

* struct ListNode {
*     int val;
*     struct ListNode *next;
* };

struct ListNode* reverseList(struct ListNode* head) {

    struct ListNode *prev = NULL, *curr = head, *next = NULL;

    while (curr != NULL) {

        next = curr->next; // Save the next node

        curr->next = prev; // Reverse the current node's next pointer
    }
}

```

```

        prev = curr;          // Move prev to current node

        curr = next;         // Move to next node
    }

    return prev; // Return the new head (previously the last node)
}

// Function to check if the linked list is a palindrome
bool isPalindrome(struct ListNode* head) {

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

        return true; // An empty list or single node list is a palindrome
    }

    // Step 1: Reverse the entire list
    struct ListNode* reversedHead = reverseList(head);

    // Step 2: Compare the original and reversed lists
    struct ListNode* original = head;

    struct ListNode* reversed = reversedHead;

    while (original != NULL && reversed != NULL) {

        if (original->val != reversed->val) {

            return false; // If any values don't match, it's not a
palindrome}

        original = original->next;

        reversed = reversed->next;}

    return true; // If all values matched, the list is a palindrome}

```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

head =
[1,2,2,1]

Output

true

Expected

true

6a) WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists.

```
#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));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

void printList(struct Node* head) {
    struct Node* temp = head;
    if (temp == NULL) {
```



```

    printf("List is empty.\n");
    return;
}
while (temp != NULL) {
    printf("%d -> ", temp->data);
    temp = temp->next;
}
printf("NULL\n");
}

void sortList(struct Node* head) {
    struct Node *i, *j;
    int temp;
    for (i = head; i != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->data > j->data) {
                // Swap data
                temp = i->data;
                i->data = j->data;
                j->data = temp;}} }}

void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
}

```

```

    }
    *head = prev;
}

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

```

```

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;
    int choice, data;

    while(1){
        printf("\nSingly Linked List Operations\n");
        printf("1. Insert in List 1\n");
        printf("2. Insert in List 2\n");
        printf("3. Display List 1\n");
        printf("4. Display List 2\n");
        printf("5. Sort List 1\n");
        printf("6. Reverse List 1\n");
        printf("7. Concatenate List 1 and List 2\n");
    }
}

```

```
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter data to insert in List 1: ");
        scanf("%d", &data);
        insertEnd(&list1, data);
        break;
    case 2:
        printf("Enter data to insert in List 2: ");
        scanf("%d", &data);
        insertEnd(&list2, data);
        break;
    case 3:
        printf("List 1: ");
        printList(list1);
        break;
    case 4:
        printf("List 2: ");
        printList(list2);
        break;
    case 5:
        sortList(list1);
        printf("List 1 after sorting: ");
        printList(list1);
        break;
    case 6:
```

```

        reverseList(&list1);

        printf("List 1 after reversing: ");
        printList(list1);

        break;

    case 7:

        concatenateLists(&list1, &list2);

        printf("List 1 after concatenation: ");
        printList(list1);

        break;

    case 8:

        printf("Exiting program...\n");

        break;

    default:

        printf("Invalid choice. Please try again.\n");

    }

}

return 0;

}

```

output:

```

Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 1
Enter data to insert in List 1: 10

```

```

Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 2
Enter data to insert in List 2: 80

```

```
Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 3
List 1: 10 -> 20 -> 30 -> NULL
```

```
Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 4
List 2: 80 -> 50 -> NULL
```

```
C:\Users\STUDENT\Desktop\Jayashree407CS\lab6a.exe
Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 5
List 1 after sorting: 10 -> 20 -> 30 -> NULL

Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 6
List 1 after reversing: 30 -> 20 -> 10 -> NULL

Singly Linked List Operations
1. Insert in List 1
2. Insert in List 2
3. Display List 1
4. Display List 2
5. Sort List 1
6. Reverse List 1
7. Concatenate List 1 and List 2
8. Exit
Enter your choice: 7
List 1 after concatenation: 30 -> 20 -> 10 -> 80 -> 50 -> NULL
```

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

Operations.

```
#include <stdio.h>

#include <stdlib.h>

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

// Stack Operations

void push(struct Node** top, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d to the stack\n", data);
}

int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack is empty. Cannot pop.\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
}

void displayStack(struct Node* top) {
    if (top == NULL) {
```

```

    printf("Stack is empty.\n");
    return;
}

struct Node* temp = top;
printf("Stack (Top -> Bottom): ");
while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
}
printf("\n");
}

// Queue Operations
void enqueue(struct Node** front, struct Node** rear, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*rear == NULL) {
        *front = *rear = newNode; // If the queue is empty, front and rear will both point to
newNode
    } else {
        (*rear)->next = newNode;
        *rear = newNode; // Move the rear to the new node
    }
    printf("Enqueued %d to the queue\n", data);
}

int dequeue(struct Node** front, struct Node** rear) {
    if (*front == NULL) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

```

```

    struct Node* temp = *front;
    int dequeuedValue = temp->data;
    *front = (*front)->next;
    if (*front == NULL) { // If the queue becomes empty after dequeue
        *rear = NULL;
    }
    free(temp);
    return dequeuedValue;
}

void displayQueue(struct Node* front) {
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue (Front -> Rear): ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* stackTop = NULL;
    struct Node* queueFront = NULL;
    struct Node* queueRear = NULL;

    int choice, data;
    do {

```



```

printf("\nMenu:\n");
printf("1. Stack Operations\n");
printf("2. Queue Operations\n");
printf("3. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1: // Stack operations
        {
            int stackChoice;
            do {
                printf("\nStack Operations:\n");
                printf("1. Push\n");
                printf("2. Pop\n");
                printf("3. Display Stack\n");
                printf("4. Back to Main Menu\n");
                printf("Enter your choice: ");
                scanf("%d", &stackChoice);

                switch (stackChoice) {
                    case 1:
                        printf("Enter data to push: ");
                        scanf("%d", &data);
                        push(&stackTop, data);
                        break;
                    case 2:
                        data = pop(&stackTop);
                        if (data != -1) {

```

```

        printf("Popped %d from the stack\n", data);
    }
    break;
case 3:
    displayStack(stackTop);
    break;
case 4:
    printf("Returning to Main Menu...\n");
    break;
default:
    printf("Invalid choice!\n");
}
} while (stackChoice != 4); // Loop until the user chooses to return to the main
menu
}
break;

case 2: // Queue operations
{
    int queueChoice;
    do {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display Queue\n");
        printf("4. Back to Main Menu\n");
        printf("Enter your choice: ");
        scanf("%d", &queueChoice);

        switch (queueChoice) {

```

```

        case 1:
            printf("Enter data to enqueue: ");
            scanf("%d", &data);
            enqueue(&queueFront, &queueRear, data);
            break;

        case 2:
            data = dequeue(&queueFront, &queueRear);
            if (data != -1) {
                printf("Dequeued %d from the queue\n", data);
            }
            break;

        case 3:
            displayQueue(queueFront);
            break;

        case 4:
            printf("Returning to Main Menu...\n");
            break;

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

    } while (queueChoice != 4); // Loop until the user chooses to return to the main
menu

    }

    break;

case 3:
    printf("Exiting program...\n");
    break;

default:
    printf("Invalid choice! Please try again.\n");
}

```

```

    } while (choice != 3); // Loop until the user chooses to exit the program

    return 0;
}

```

output:

```

C:\Users\STUDENT\Desktop\Jayashree407CS\lab6b.exe
Menu:
1. Stack Operations
2. Queue Operations
3. Exit
Enter your choice: 1

Stack Operations:
1. Push
2. Pop
3. Display Stack
4. Back to Main Menu
Enter your choice: 1
Enter data to push: 10
Pushed 10 to the stack

Stack Operations:
1. Push
2. Pop
3. Display Stack
4. Back to Main Menu
Enter your choice: 1
Enter data to push: 20
Pushed 20 to the stack

Stack Operations:
1. Push
2. Pop
3. Display Stack
4. Back to Main Menu
Enter your choice: 1
Enter data to push: 30
Pushed 30 to the stack

Stack Operations:
1. Push
2. Pop
3. Display Stack
4. Back to Main Menu
Enter your choice: 3
Stack (Top -> Bottom): 30 20 10

```

```

C:\Users\STUDENT\Desktop\Jayashree407CS\lab6b.exe
3. Display Queue
4. Back to Main Menu
Enter your choice: 1
Enter data to enqueue: 6
Enqueued 6 to the queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Back to Main Menu
Enter your choice: 1
Enter data to enqueue: 70
Enqueued 70 to the queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Back to Main Menu
Enter your choice: 3
Queue (Front -> Rear): 50 6 70

Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Back to Main Menu
Enter your choice: 2
Dequeued 50 from the queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Back to Main Menu
Enter your choice: 3
Queue (Front -> Rear): 6 70

```

Lab program 7

7a) 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

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

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

```
// Define the structure for a Node
```

```
struct Node {
```

```
    int data;          // Holds the value of the node
```

```
    struct Node* prev; // Points to the previous node
```

```
    struct Node* next; // Points to the next node
```

```
};
```

```
// Declare the head of the list globally
```

```
struct Node* head = NULL;
```

```
// Function to create a new node
```

```
struct Node* createNode(int value) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    if (newNode == NULL) {
```

```
        printf("Memory allocation failed.\n");
```

```
        return NULL;
```

```
    }
```

```
    newNode->data = value;
```

```
    newNode->prev = newNode->next = NULL;
```

```
    return newNode;
```

```
}
```

```

// Function to create a doubly linked list
void createList() {
    int value, choice;
    struct Node* temp;

    do {
        // Prompt user to enter a value
        printf("Enter value to insert: ");
        scanf("%d", &value);

        // Create a new node with the entered value
        struct Node* newNode = createNode(value);
        if (head == NULL) {
            head = newNode; // If the list is empty, the new node becomes the head
        } else {
            temp = head;
            // Traverse to the last node
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode; // Update the last node's next pointer
            newNode->prev = temp; // Set the new node's previous pointer to the last node
        }

        // Ask user if they want to add another node
        printf("Do you want to add another node? (1 for Yes, 0 for No): ");
        scanf("%d", &choice);
    } while (choice != 0);
}

```

```

// Function to insert a new node to the left of a specific node
void insertLeft(int value, int target) {
    struct Node* temp = head;

    // Search for the node with the target value
    while (temp != NULL && temp->data != target) {
        temp = temp->next; // Traverse the list to find the target node
    }
    if (temp == NULL) {
        printf("Node with value %d not found.\n", target);
        return;
    }
    // Create a new node to insert
    struct Node* newNode = createNode(value);

    // Insert the new node to the left of the target node
    newNode->next = temp;
    newNode->prev = temp->prev;
    // Update the previous node's next pointer, if it exists
    if (temp->prev != NULL) {
        temp->prev->next = newNode;
    } else {
        head = newNode; // If inserting at the head, update the head pointer
    }

    // Update the target node's previous pointer
    temp->prev = newNode;
    printf("Node with value %d inserted to the left of %d.\n", value, target);
}

```

```

// Function to delete a node based on a specific value
void deleteNode(int value) {
    struct Node* temp = head;

    // Search for the node to delete
    while (temp != NULL && temp->data != value) {
        temp = temp->next; // Traverse to find the node with the given value
    }

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

    // If the node has a previous node, update its next pointer
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        head = temp->next; // If deleting the head, update the head pointer
    }

    // If the node has a next node, update its previous pointer
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    free(temp); // Free the memory of the deleted node
    printf("Node with value %d deleted.\n", value);
}

// Function to display the contents of the doubly linked list
void displayList() {
    if (head == NULL) {
        printf("List is empty.\n");
    }
}

```



```

        return;
    }

    struct Node* temp = head;
    // Traverse the list and print each node's data
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function with a menu-driven approach
int main() {
    int choice, value, target;

    do {
        // Display the menu
        printf("\nDoubly Linked List Operations:\n");
        printf("1. Create a doubly linked list\n");
        printf("2. Insert a new node to the left of a specific node\n");
        printf("3. Delete a node based on specific value\n");
        printf("4. Display the contents of the list\n");
        printf("5. Exit\n");

        // Ask user for their choice
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:

```

```

        createList(); // Create a new doubly linked list
        break;
    case 2:
        // Ask user for the value to insert and the target node
        printf("Enter value to insert: ");
        scanf("%d", &value);
        printf("Enter target value (left of which node to insert): ");
        scanf("%d", &target);
        insertLeft(value, target); // Insert the node to the left of the target node
        break;
    case 3:
        // Ask user for the value of the node to delete
        printf("Enter value to delete: ");
        scanf("%d", &value);
        deleteNode(value); // Delete the node with the specified value
        break;
    case 4:
        displayList(); // Display the contents of the list
        break;
    case 5:
        printf("Exiting the program.\n");
        break;
    default:
        printf("Invalid choice. Please try again.\n");
}
} while (choice != 5); // Repeat until the user chooses to exit

return 0;
}

```

output:

```
C:\Users\STUDENT\Desktop\Jayashree407CS\lab7.exe

Doubly Linked List Operations:
1. Create a doubly linked list
2. Insert a new node to the left of a specific node
3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 1
Enter value to insert: 10
Do you want to add another node? (1 for Yes, 0 for No): 1
Enter value to insert: 20
Do you want to add another node? (1 for Yes, 0 for No): 1
Enter value to insert: 30
Do you want to add another node? (1 for Yes, 0 for No): 0

Doubly Linked List Operations:
1. Create a doubly linked list
2. Insert a new node to the left of a specific node
3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 2
Enter value to insert: 15
Enter target value (left of which node to insert): 20
Node with value 15 inserted to the left of 20.

Doubly Linked List Operations:
1. Create a doubly linked list
2. Insert a new node to the left of a specific node
3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 4
10 <-> 15 <-> 20 <-> 30 <-> NULL
```

```
C:\Users\STUDENT\Desktop\Jayashree407CS\lab7.exe

3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 4
10 <-> 15 <-> 20 <-> 30 <-> NULL

Doubly Linked List Operations:
1. Create a doubly linked list
2. Insert a new node to the left of a specific node
3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 3
Enter value to delete: 20
Node with value 20 deleted.

Doubly Linked List Operations:
1. Create a doubly linked list
2. Insert a new node to the left of a specific node
3. Delete a node based on specific value
4. Display the contents of the list
5. Exit
Enter your choice: 4
10 <-> 15 <-> 30 <-> NULL
```

Lab Program 8

8a) 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.

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

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

```
// Structure for a binary tree node
```

```
struct Node {
```

```
    int data;
```

```
    struct Node* left;
```

```
    struct Node* right;
```

```
};
```

```
// Function to create a new node with given data
```

```
struct Node* createNode(int data) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;
```

```
    newNode->left = NULL;
```

```
    newNode->right = NULL;
```

```
    return newNode;
```

```
}
```

```
// Function to insert a new node in the binary search tree
```

```
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;
}

// In-order traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Pre-order traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);

```

```

        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

```

// Function to display the menu and perform actions based on user input

```

void displayMenu() {
    printf("\nBinary Search Tree Operations:\n");
    printf("1. Insert a node\n");
    printf("2. In-order traversal\n");
    printf("3. Pre-order traversal\n");
    printf("4. Post-order traversal\n");
    printf("5. Exit\n");
}

int main() {
    struct Node* root = NULL; // Initialize an empty tree
    int choice, data;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Insert a node
                printf("Enter value to insert: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2: // In-order traversal

```

```

        printf("In-order traversal: ");
        inorderTraversal(root);
        printf("\n");
        break;
case 3: // Pre-order traversal
        printf("Pre-order traversal: ");
        preorderTraversal(root);
        printf("\n");
        break;
case 4: // Post-order traversal
        printf("Post-order traversal: ");
        postorderTraversal(root);
        printf("\n");
        break;
case 5: // Exit
        printf("Exiting the program.\n");
        exit(0);

default:
        printf("Invalid choice, please try again.\n");} }
return 0;
}

```

output:

C:\Users\STUDENT\Desktop\Jayashree407CS\lab8.exe

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 1

Enter value to insert: 10

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 1

Enter value to insert: 20

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 1

Enter value to insert: 30

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 1

Enter value to insert: 50

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 2

In-order traversal: 10 20 30 50

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 3

Pre-order traversal: 10 20 30 50

Binary Search Tree Operations:

1. Insert a node
2. In-order traversal
3. Pre-order traversal
4. Post-order traversal
5. Exit

Enter your choice: 4

Post-order traversal: 50 30 20 10

Lab program 9

9a) Write a program to traverse a graph using BFS method.

```
#include <stdio.h>

#define MAX 5

void bfs(int adj[][MAX], int visited[], int start) {
    int q[MAX], front = -1, rear = -1, i;
    for (i = 0; i < MAX; i++)
        visited[i] = 0;
    q[++rear] = start;
    ++front;
    visited[start] = 1;
    while (rear >= front) {
        start = q[front++];
        printf("%c -> ", start + 'A');
        for (i = 0; i < MAX; i++) {
            if (adj[start][i] && visited[i] == 0) {
                q[++rear] = i;
                visited[i] = 1;
            }
        }
        printf("\n");
    }
}

int main() {
    int adj[MAX][MAX], visited[MAX], i, j;
    printf("Enter the adjacency matrix\n");
    for (i = 0; i < MAX; i++) {
        for (j = 0; j < MAX; j++) {
            scanf("%d", &adj[i][j]);
        }
    }
}
```

```

printf("\nBFS\n");
bfs(adj, visited, 0);
return 0;
}

```

output:

```

Enter the adjacency matrix
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0

BFS
A -> B -> C -> D -> E ->

```

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

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

```
#define MAX 5 // Maximum number of vertices in the graph
```

```
// Function to perform DFS traversal
```

```
void dfs(int adj[][MAX], int visited[], int start) {
```

```
    int s[MAX], top = -1, i;
```

```
    // Push the starting vertex onto the stack and mark it as visited
```

```
    s[++top] = start;
```

```
    visited[start] = 1;
```

```
    // Perform DFS traversal
```

```
    while (top != -1) {
```

```
        // Pop the top element from the stack

```

```

start = s[top--];

printf("%c -> ", start + 'A'); // Print the current vertex


// Traverse all neighbors of the current vertex
for (i = 0; i < MAX; i++) {
    if (adj[start][i] && visited[i] == 0) { // If there's an edge and the vertex is unvisited
        s[++top] = i; // Push the vertex onto the stack
        visited[i] = 1; // Mark it as visited
    }
}
}

}

int isConnected(int adj[][MAX]) {
    int visited[MAX] = {0}; // Initialize all vertices as unvisited
    int i;

    // Perform DFS starting from vertex 0
    dfs(adj, visited, 0);

    // Check if all vertices are visited
    for (i = 0; i < MAX; i++) {
        if (visited[i] == 0) {
            return 0; // Graph is not connected
        }
    }

    return 1; // Graph is connected
}

```

```

int main() {
    int adj[MAX][MAX], i, j;

    // Input the adjacency matrix
    printf("Enter the adjacency matrix (5x5):\n");
    for (i = 0; i < MAX; i++) {
        for (j = 0; j < MAX; j++) {
            scanf("%d", &adj[i][j]);
        }
    }

    // Check if the graph is connected
    if (isConnected(adj)) {
        printf("\nThe graph is connected.\n");
    } else {
        printf("\nThe graph is not connected.\n");
    }

    return 0;
}

```

output:

```

Enter the adjacency matrix (5x5):
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0
A -> C -> E -> D -> B ->
The graph is connected.

```

```

=== Code Execution Successful ===

```

Lab Program 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 m 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 $H: K \rightarrow L$ as $H(K) = K \bmod m$ (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.

```
#include <stdio.h>

#include <stdlib.h>

int key[20], n, m;

int *ht;

int count = 0;

void insert(int key) {
    int index = key % m;
    while (ht[index] != -1) {
        index = (index + 1) % m;
    }
    ht[index] = key;
    count++;
}

void display() {
    int i;
    if (count == 0) {
        printf("\nHash Table is empty");
        return;
    }
    printf("\nHash Table contents are:\n");
    for (i = 0; i < m; i++) {
        printf("\n T[%d] --> %d", i, ht[i]);
    }
}
```

```

}

int main() {
    int i;

    printf("\nEnter the number of employee records (N): ");
    scanf("%d", &n);

    printf("\nEnter the memory size (m) for the hash table: ");
    scanf("%d", &m);

    ht = (int *)malloc(m * sizeof(int));

    if (!ht) {
        printf("\nMemory allocation failed.");
        return 1;
    }

    for (i = 0; i < m; i++) {
        ht[i] = -1;
    }

    printf("\nEnter the four-digit key values (K) for %d Employee Records:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &key[i]);
    }

    for (i = 0; i < n; i++) {
        if (count == m) {
            printf("\nHash table is full. Cannot insert the record for key %d", key[i]);
            break;
        }
        insert(key[i]);
    }

    display();

    return 0;
}

```

output:

```
Enter the number of employee records (N): 4
```

```
Enter the memory size (m) for the hash table: 5
```

```
Enter the four-digit key values (K) for 4 Employee Records:  
1234 5678 9123 4567 2345
```

```
Hash Table contents are:
```

```
T[0] --> 9123
```

```
T[1] --> -1
```

```
T[2] --> 4567
```

```
T[3] --> 5678
```

```
T[4] --> 1234
```