

**Data structures and Algorithms Lab Assessment - 1**

# PROGRAM 1

### Stack

#include<stdio.h> struct stack{

int arr[5]; int top;

}st;

void st\_push(){ int item;

if(st.top == 4){ printf("Stack full");

}

else{

printf("Enter Value: "); scanf("%d",&item); st.arr[++st.top] = item;

}

}

void st\_pop(){ if(st.top == -1){

printf("Stack Empty");

}

else{

st.top--;

}

}

void st\_display(){ int temp; if(st.top == -1){

printf("Stack Empty");

}

else{

temp = st.top; while(temp>=0){

printf("%d ", st.arr[temp]); temp--;

}

}

}

void main(){

int ch; st.top = -1;

do{

printf("\n1.push\n2.pop\n3.display\n4.exit\n"); printf("Enter your choice : ");

scanf("%d", &ch);

switch(ch){

case 1:

st\_push(); break;

case 2:

st\_pop(); break;

case 3:

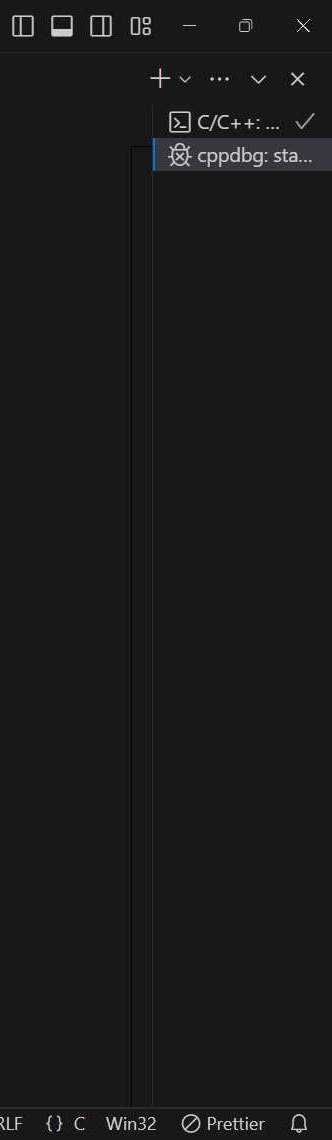
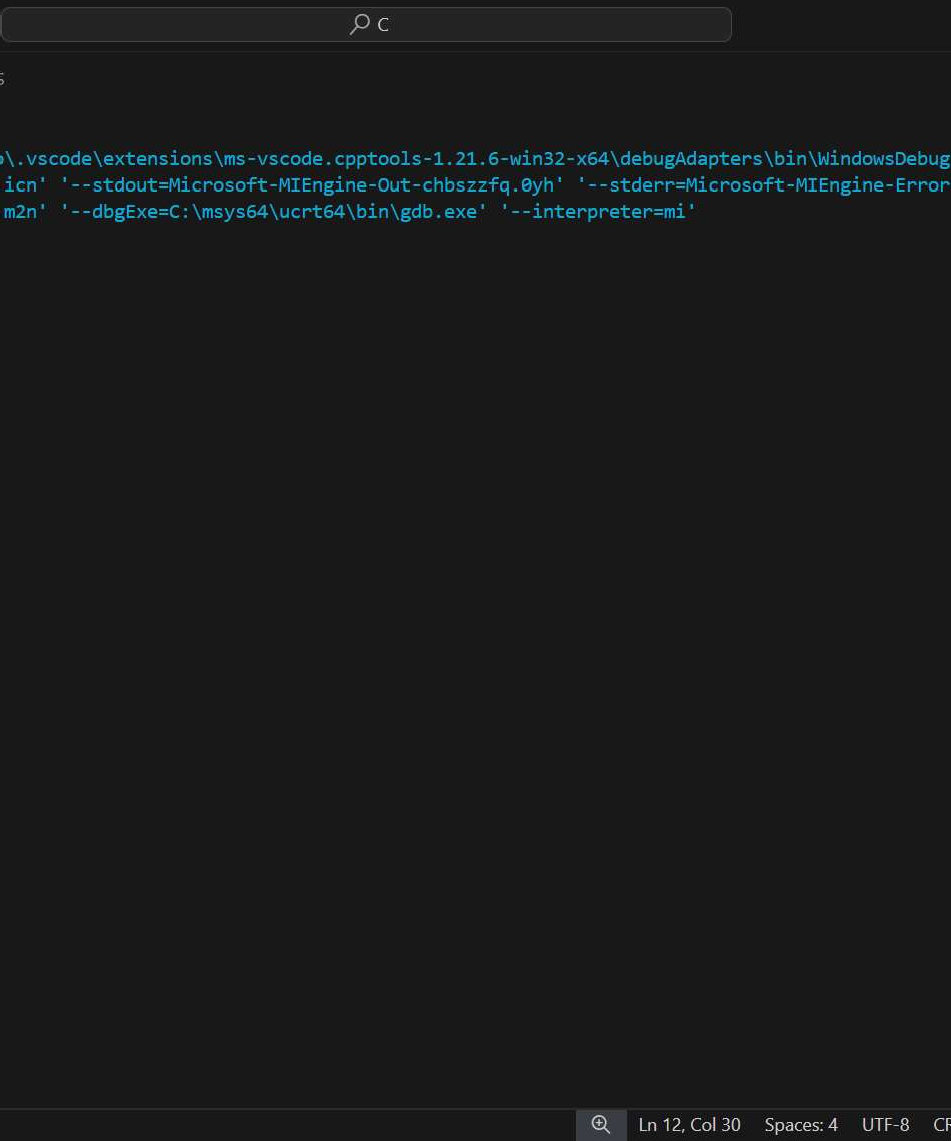
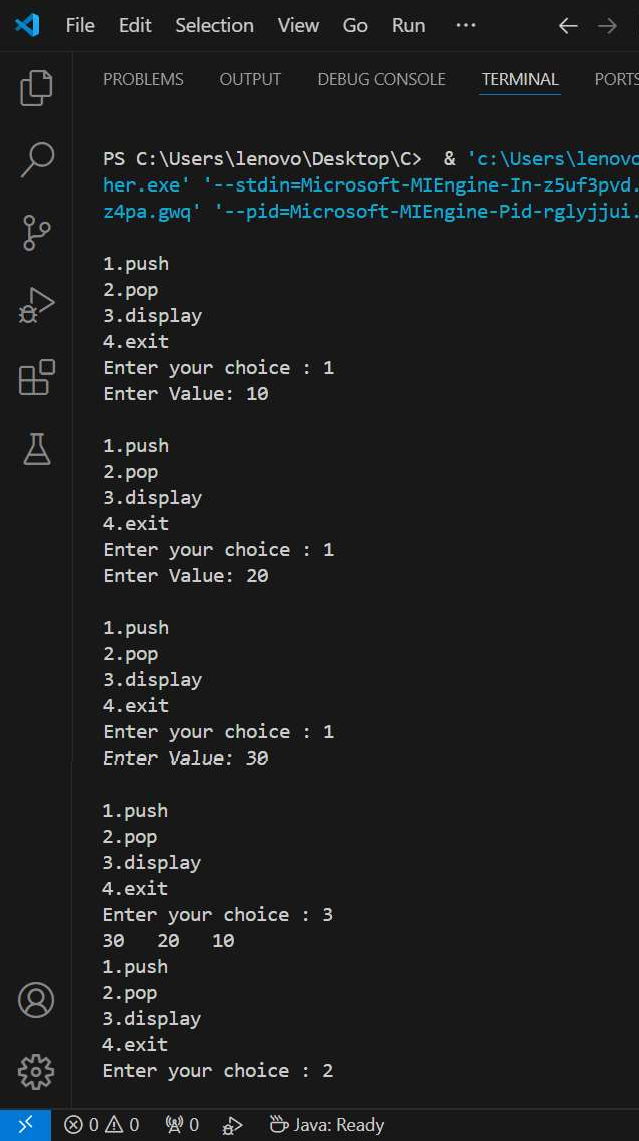
st\_display(); break;

}

}while(ch<4);

}

**OUTPUT:**



# PROGRAM 2

**Infix to Postfix**

#include <stdio.h>

// Stack structure struct Stack {

char elements[20]; int top;

} operatorStack;

// Arrays for infix and char postfixExpr[20]; char infixExpr[20];

postfix expressions

// Function to push a value onto the stack void push(char value) {

operatorStack.elements[++operatorStack.top] = value;

}

// Function to pop a value from the stack char pop() {

return operatorStack.elements[operatorStack.top--];

}

// Function to check the precedence of operators int getPrecedence(char operator) {

if (operator == '/' || operator == '\*') return 2; if (operator == '+' || operator == '-') return 1; return 0;

}

// Main function to convert infix to postfix void main() {

int postfixIndex = -1; operatorStack.top = -1; int i, precedence;

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

for (i = 0; infixExpr[i] != '\0'; i++) { precedence = getPrecedence(infixExpr[i]);

if (precedence != 0) {

while (operatorStack.top != -1 && precedence <= getPrecedence(operatorStack.elements[operatorStack.top])) {

postfixExpr[++postfixIndex] = pop();

}

push(infixExpr[i]);

} else if (infixExpr[i] == '(') { push(infixExpr[i]);

} else if (infixExpr[i] == ')') { while (operatorStack.top != -1

&&operatorStack.elements[operatorStack.top] != '(') {

postfixExpr[++postfixIndex] = pop();

}

pop(); // Remove '(' from stack

} else {

postfixExpr[++postfixIndex] = infixExpr[i];

}

}

// Pop all remaining operators from the stack while (operatorStack.top != -1) {

postfixExpr[++postfixIndex] = pop();

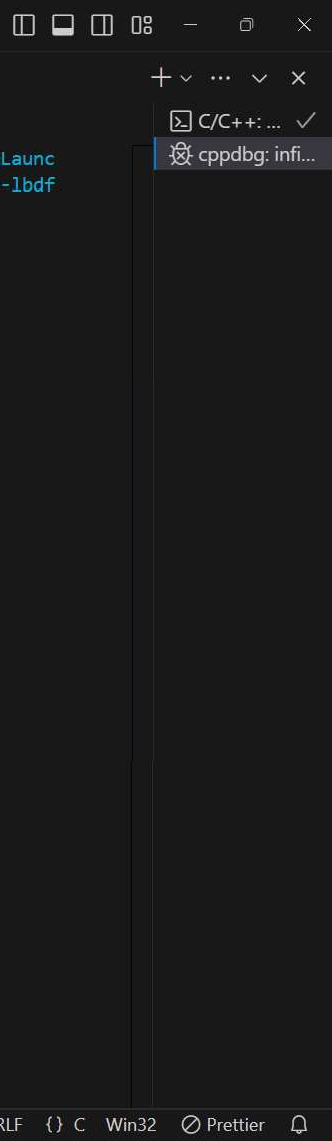
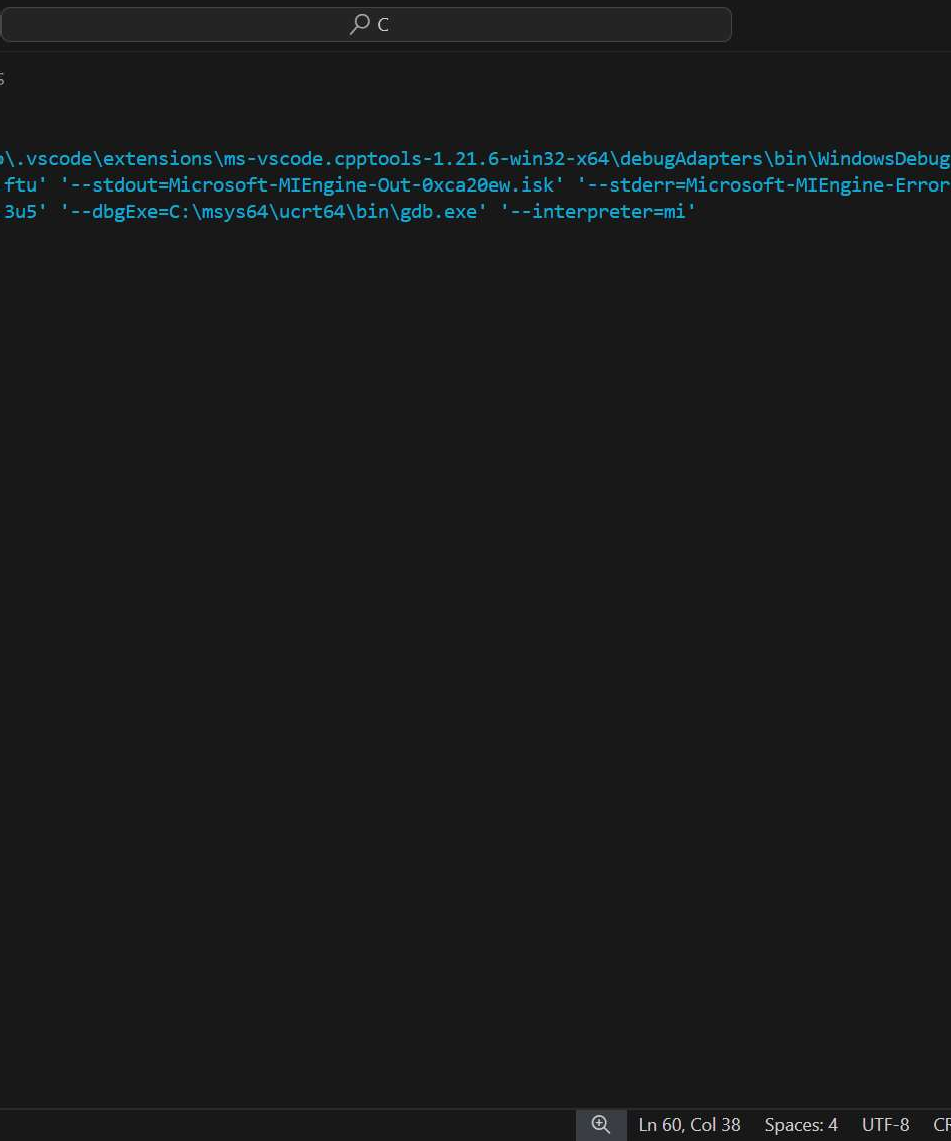
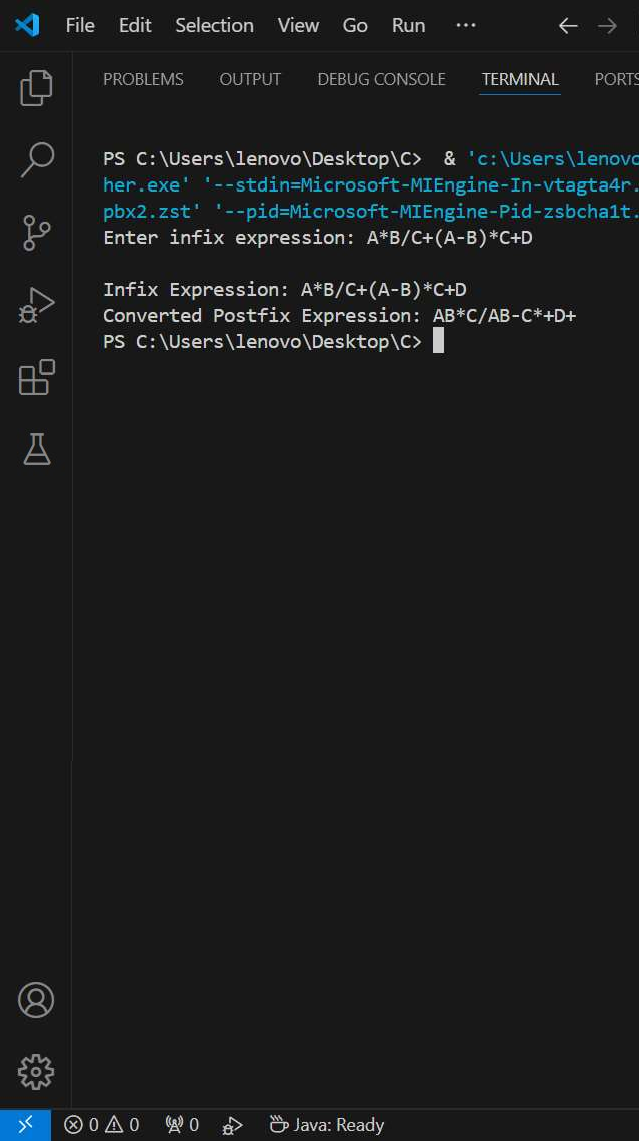
}

postfixExpr[++postfixIndex] = '\0'; // Null-terminate the postfix expression

printf("\nInfix Expression: %s", infixExpr); printf("\nConverted Postfix Expression: %s\n", postfixExpr);

}

**OUTPUT:**



# PROGRAM 3

### Infix to Prefix

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

// Stack structure to hold operators struct Stack {

char elements[15]; int top;

} operatorStack;

// Arrays for prefix and char prefixExpr[20]; char infixExpr[20];

infix expressions

// Function to push a value onto the stack void push(char value) {

operatorStack.elements[++operatorStack.top] = value;

}

// Function to pop a value from the stack

char pop() {

return operatorStack.elements[operatorStack.top--];

}

// Function to check the precedence of operators int getPrecedence(char operator) {

if (operator == '/' || operator == '\*') return 2; if (operator == '+' || operator == '-') return 1; return 0;

}

// Function to reverse a string void reverseString(char \*exp) {

int length = strlen(exp);

for (int i = 0; i< length / 2; i++) { char temp = exp[i];

exp[i] = exp[length - i - 1]; exp[length - i - 1] = temp;

}

}

// Function to replace parentheses void replaceParentheses(char \*exp) {

for (int i = 0; exp[i] != '\0'; i++) { if (exp[i] == '(')

exp[i] = ')';

else if (exp[i] == ')')

exp[i] = '(';

}

}

// Function to convert infix expression to prefix void convertInfixToPrefix() {

int prefixIndex = -1; operatorStack.top = -1;

int i, precedence;

// Step 1: Reverse the infix expression reverseString(infixExpr);

// Step 2: Replace parentheses replaceParentheses(infixExpr);

// Step 3: Convert to prefix using similar logic to infix to postfix for (i = 0; infixExpr[i] != '\0'; i++) {

precedence = getPrecedence(infixExpr[i]);

if (precedence != 0) {

while (operatorStack.top != -1 && precedence

<getPrecedence(operatorStack.elements[operatorStack.top])) { prefixExpr[++prefixIndex] = pop();

}

push(infixExpr[i]);

} else if (infixExpr[i] == '(') {

push(infixExpr[i]);

} else if (infixExpr[i] == ')') { while (operatorStack.top != -1

&&operatorStack.elements[operatorStack.top] != '(') { prefixExpr[++prefixIndex] = pop();

}

pop(); // Remove '(' from stack

} else { prefixExpr[++prefixIndex] = infixExpr[i];

}

}

// Pop all remaining

operators from the stack

while (operatorStack.top != -1) { prefixExpr[++prefixIndex] = pop();

}

prefixExpr[++prefixIndex] = '\0';

// Step 4: Reverse the result to get the final prefix expression reverseString(prefixExpr);

}

int main() {

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

convertInfixToPrefix();

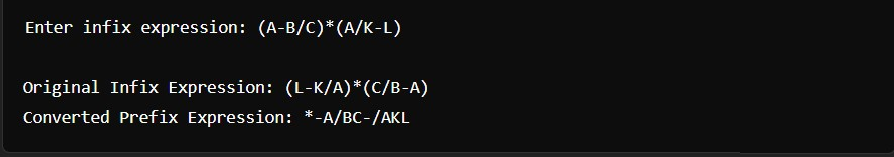
printf("\nOriginal Infix Expression: %s", infixExpr);

printf("\nConverted Prefix Expression: %s\n", prefixExpr);

return 0;

}

**OUTPUT:**



# PROGRAM 4

### Postfix Evaluation

#include <stdio.h> #include <ctype.h>

struct Stack {

int data[20]; int index;

} numStack;

void push(int num) { numStack.data[++numStack.index] = num;

}

int pop() {

return numStack.data[numStack.index--];

}

int evaluatePostfix(char\* expression) { numStack.index = -1;

for (int i = 0; expression[i] != '\0'; i++) { if (isdigit(expression[i])) {

push(expression[i] - '0');

} else {

int val2 = pop(); int val1 = pop(); int result;

switch (expression[i]) { case '+':

result = val1 + val2; break;

case '-':

result = val1 - val2; break;

case '\*':

result = val1 \* val2; break;

case '/':

result = val1 / val2; break;

}

push(result);

}

}

return pop();

}

int main() {

char expression[20]; printf("Enter postfix expression: "); scanf("%s", expression);

int finalResult = evaluatePostfix(expression); printf("Result: %d\n", finalResult);

return 0;

}



# PROGRAM 5

**Evaluate prefix**

#include <stdio.h> #include <ctype.h> #include <string.h>

struct Stack {

int elements[20]; int topIndex;

} operandStack;

void push(int value) { operandStack.elements[++operandStack.topIndex] = value;

}

int pop() {

return operandStack.elements[operandStack.topIndex--];

}

int evaluatePrefix(char\* operandStack.topIndex =

expr) {

-1;

int length = strlen(expr);

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

if (isdigit(expr[i])) {

push(expr[i]

} else {

int operand1 int operand2 int result;

- '0');

= pop();

= pop();

switch (expr[i]) { case '+':

result = operand1 + operand2; break;

case '-':

result = operand1 - operand2; break;

case '\*':

result = operand1 \* operand2; break;

case '/':

result = operand1 / operand2; break;

}

push(result);

}

}

return pop();

}

int main() {

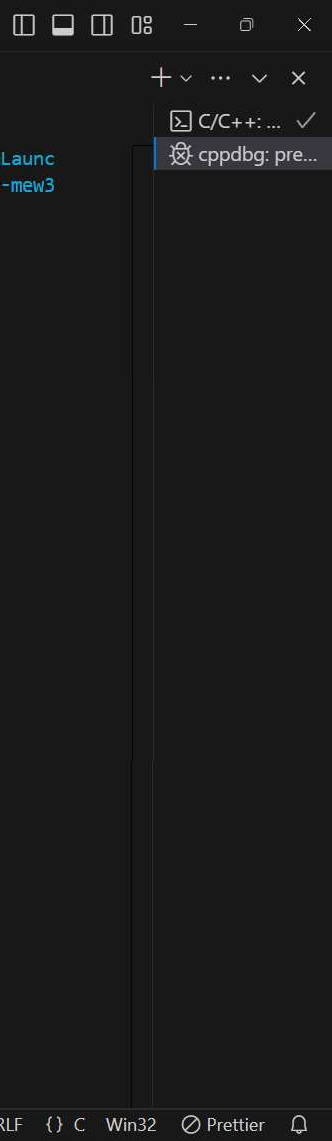
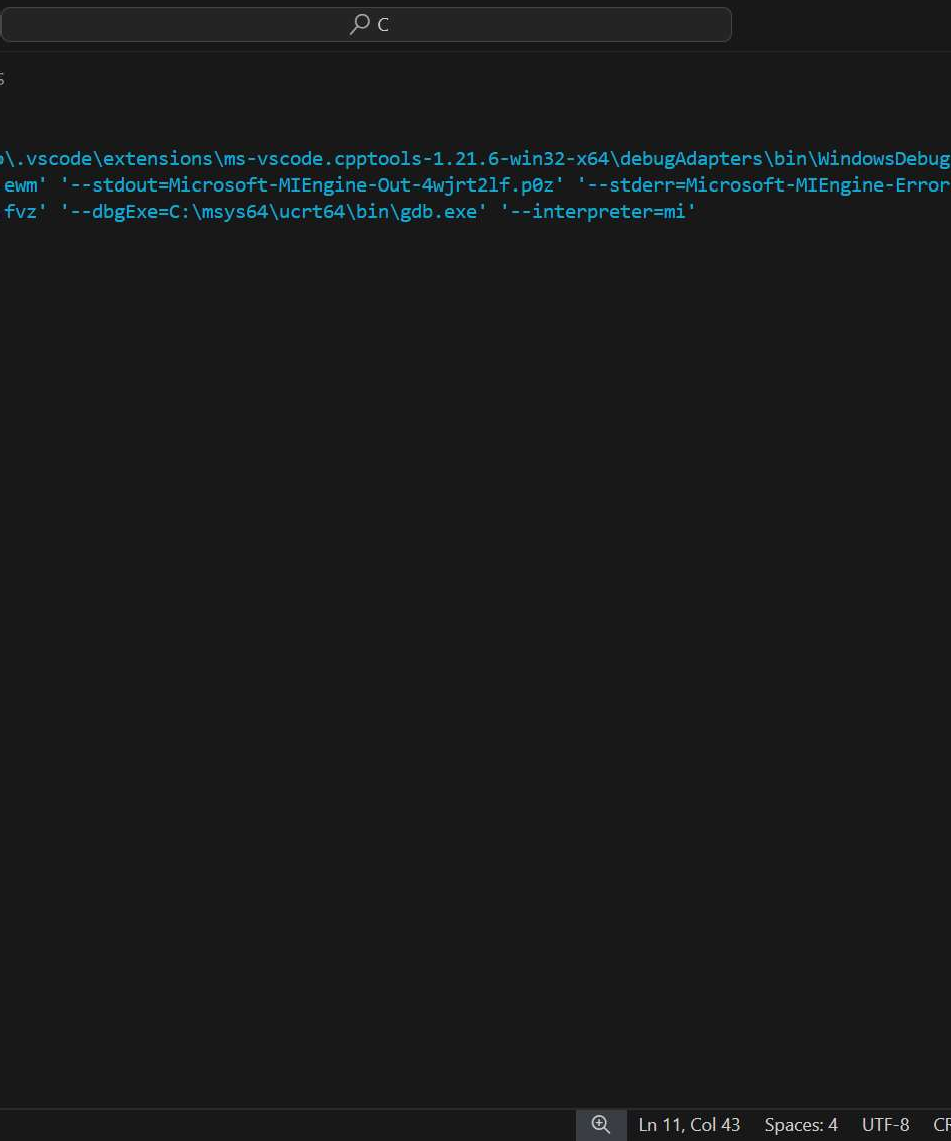
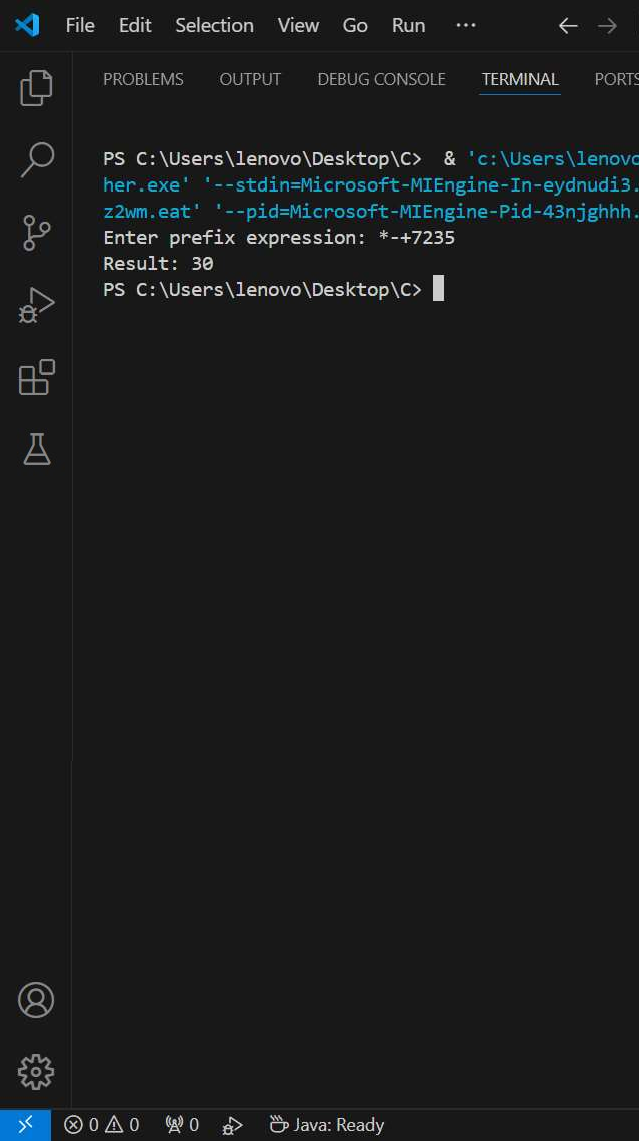
char expr[20];

printf("Enter prefix expression: "); scanf("%s", expr);

int result = evaluatePrefix(expr); printf("Result: %d\n", result);

return 0;

}



# PROGRAM 6

### Queue

#include <stdio.h> struct Queue {

int items[5]; int start, end;

} myQueue;

void enqueue() { int value;

if (myQueue.end == 4) {

printf("Queue is full\n");

} else {

printf("Enter value to enqueue: "); scanf("%d", &value); myQueue.items[++myQueue.end] = value;

if (myQueue.end == 0) { myQueue.start = 0;

}

}

}

void dequeue() {

if (myQueue.start == -1) { printf("Queue is empty\n");

} else {

printf("Dequeued: %d\n", myQueue.items[myQueue.start]); if (myQueue.start == myQueue.end) {

myQueue.start = -1;

myQueue.end = -1;

} else { myQueue.start++;

}

}

}

void display() {

if (myQueue.start == -1) { printf("Queue is empty\n");

} else {

printf("Queue contents: ");

for (int i = myQueue.start; i<= myQueue.end; i++) { printf("%d ", myQueue.items[i]);

}

printf("\n");

}

}

int main() { myQueue.start = -1;

myQueue.end = -1; int option;

do {

printf("1) Enqueue 2) Dequeue 3) Display 4) Quit\n"); printf("Enter your choice: ");

scanf("%d", &option);

switch (option) { case 1:

enqueue(); dequeue();

break; case 2:

break;

display();

case 3:

break; case 4:

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

break; default:

printf("Invalid choice.

break;

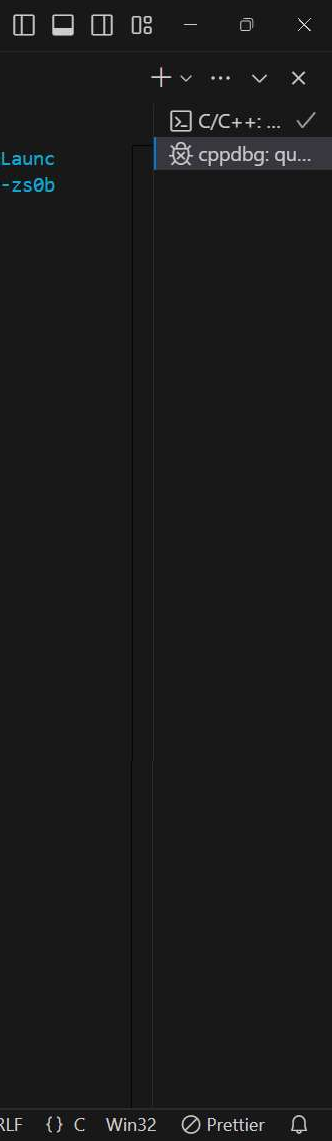
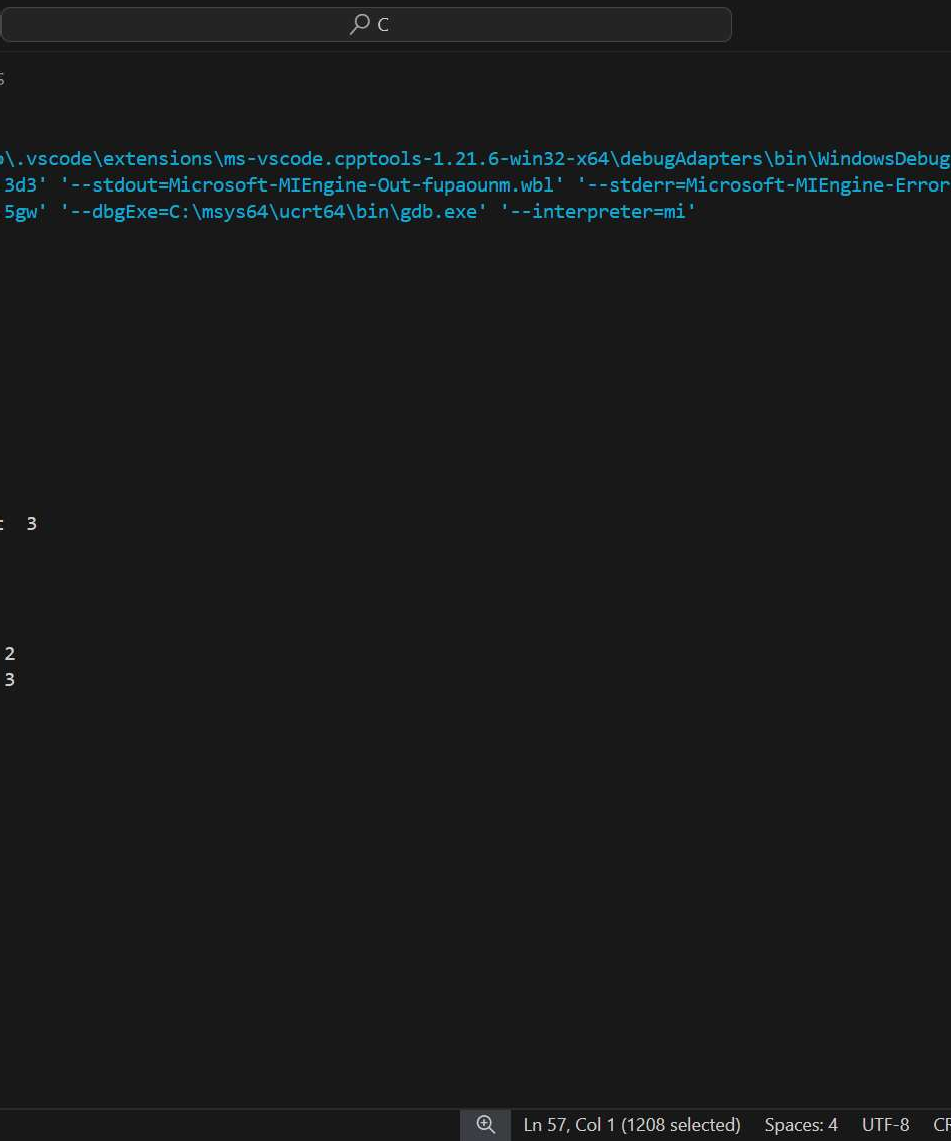
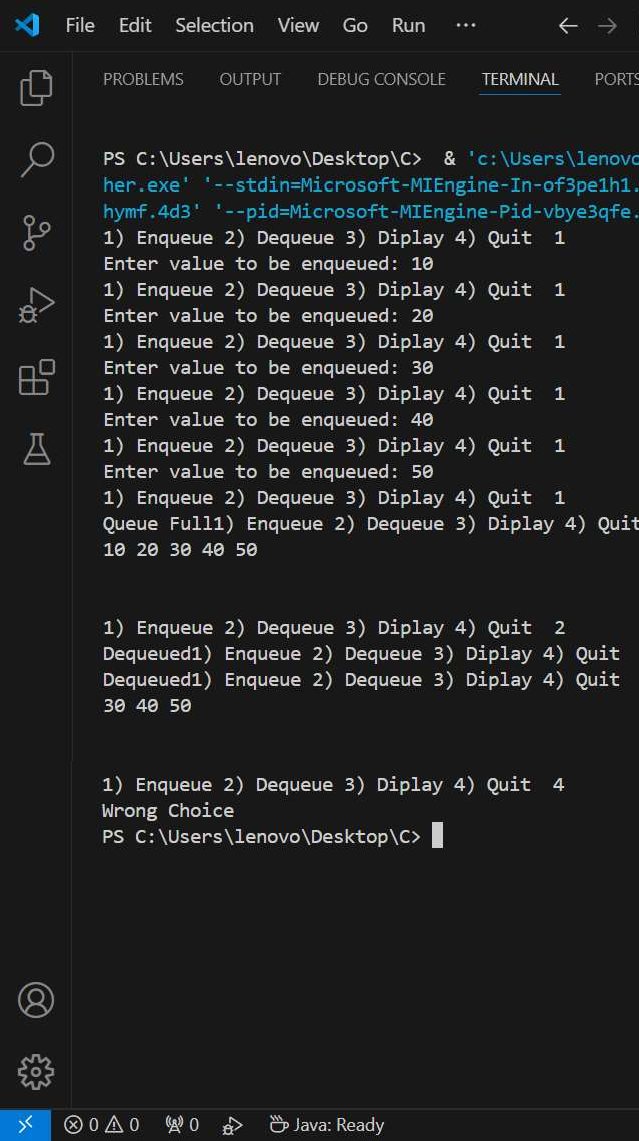
Please enter a number between 1 and 4.\n");

}

} while (option != 4);

return 0;

}



# PROGRAM 7

### Circular Queue

#include <stdio.h>

struct CircularQueue { int buffer[5];

int head, tail;

} queue;

void enqueue() { int value;

if ((queue.tail + 1)

% 5 == queue.head) {

printf("Queue is full\n");

} else {

printf("Enter value to enqueue: "); scanf("%d", &value);

queue.tail = (queue.tail + 1) % 5; queue.buffer[queue.tail] = value;

if (queue.head == -1) { queue.head = queue.tail;

}

printf("Enqueued: %d\n\n", value);

}

}

void dequeue() {

if (queue.head == -1) { printf("Queue is empty\n");

} else {

printf("Dequeued: %d\n", queue.buffer[queue.head]); if (queue.head == queue.tail) {

queue.head = -1;

queue.tail = -1;

} else {

queue.head = (queue.head + 1) % 5;

}

}

}

void display() {

if (queue.head == -1) { printf("Queue is empty\n");

} else {

int i = queue.head; printf("Queue contents: ");

while (1) {

printf("%d ", queue.buffer[i]);

if (i == queue.tail) break; i = (i + 1) % 5;

}

printf("\n\n");

}

}

int main() { queue.head = -1;

queue.tail = -1; int option;

do {

printf("1) Enqueue 2) Dequeue 3) Display 4) Quit\n"); printf("Enter your choice: ");

scanf("%d", &option);

switch (option) { case 1:

enqueue(); dequeue();

break; case 2:

display();

break; case 3:

break; case 4:

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

break; default:

printf("Invalid choice.

break;

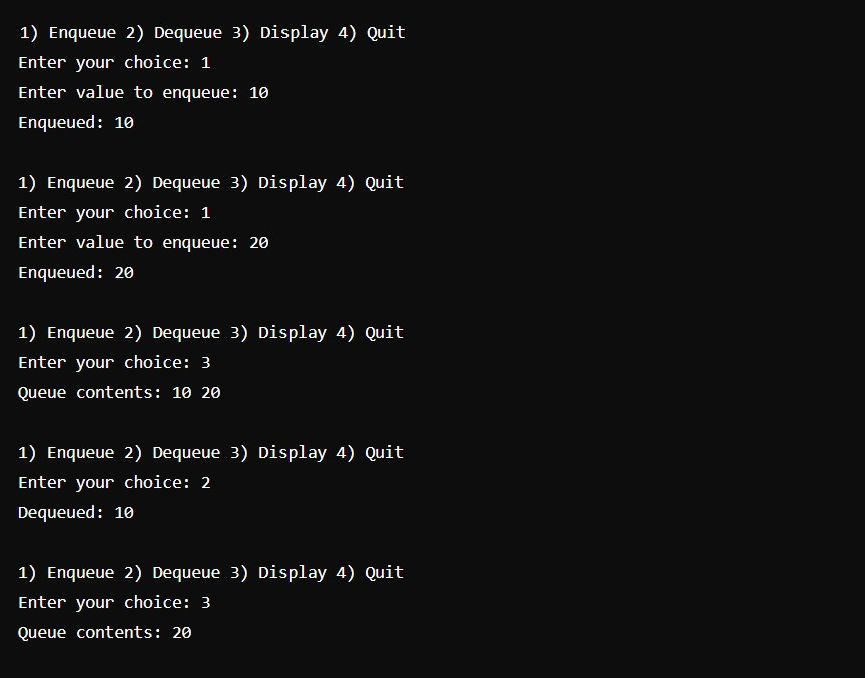
Please enter a number between 1 and 4.\n");

}

} while (option != 4);

return 0;

}



## PROGRAM-8

### Priority Queue

#include <stdio.h> struct PriorityQueue {

int a[5]; int size;

} q;

void enqueue(){ int val, i, j;

if(q.size == 5) { printf("Queue Full\n");

} else {

printf("Enter value to be enqueued: "); scanf("%d", &val);

// Insert while maintaining the priority (min-priority queue) if(q.size == 0) {

q.a[0] = val;

} else {

for(i = q.size - 1; i >= 0 && q.a[i] > val; i--) { q.a[i + 1] = q.a[i];

}

q.a[i + 1] = val;

}

q.size++; printf("Enqueued\n\n");

}

}

void dequeue(){ if(q.size == 0) {

printf("Queue Empty\n");

} else {

printf("Dequeued: %d\n", q.a[0]);

// Shift elements to the left after dequeuing for(int i = 0; i < q.size - 1; i++) {

q.a[i] = q.a[i + 1];

}

q.size--;

}

}

void display(){ if(q.size == 0) {

printf("Queue Empty\n");

} else {

for(int i = 0; i < q.size; i++) { printf("%d ", q.a[i]);

}

printf("\n\n\n");

}

}

void main(){

q.size = 0; int choice; do{

printf("1) Enqueue 2) Dequeue 3) Display 4) Quit "); scanf("%d", &choice);

switch (choice)

{

case 1:

enqueue(); break;

case 2:

dequeue(); break;

case 3:

display(); break;

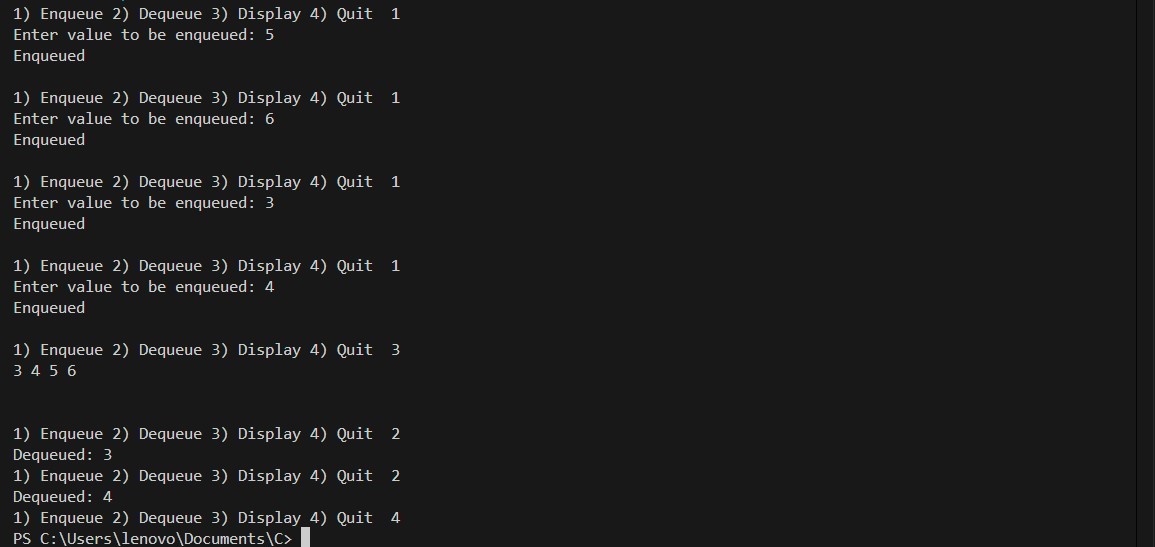
default:

if (choice != 4) printf("Wrong Choice\n");

}

}while (choice != 4);

}



## PROGRAM-9

### Linked List

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

struct node {

int data;

struct node \*next;

struct node \*head = NULL, \*tail = NULL, \*temp; int len = 0;

// Function to insert a node at the end void insert() {

struct node \*newNode = (struct node \*)malloc(sizeof(struct node)); printf("Enter integer to be inserted: ");

scanf("%d", &newNode->data); newNode->next = NULL;

if(head == NULL) { // If it's the first node head = newNode;

} else { // If list already exists tail->next = newNode;

}

tail = newNode; len++;

printf("\nNode Inserted\n");

}

// Function to insert a node at the beginning

void insertAtBeginning() { struct node \*newNode = (struct node

\*)malloc(sizeof(struct node));

printf("Enter integer to be inserted: "); scanf("%d", &newNode->data);

if(head == NULL) { tail = newNode;

}

newNode->next = head; head = newNode; len++;

printf("\nNode Inserted\n");

}

// Function to insert a node at a specific position void insertAtPos() {

int pos;

printf("Enter position: "); scanf("%d", &pos);

if(pos > len + 1 || pos < 1) { printf("Wrong position entered.\n"); return;

}

if(pos == 1) { insertAtBeginning(); return;

}

if(pos == len + 1) { insert(); return;

}

struct node \*newNode = (struct node \*)malloc(sizeof(struct node)); printf("Enter integer to be inserted: ");

scanf("%d", &newNode->data);

temp = head;

while (--pos != 1) { temp = temp->next;

}

newNode->next = temp->next; temp->next = newNode; len++;

printf("\nNode Inserted\n");

}

// Function to display the linked list void display() {

if(head == NULL) { printf("Empty List\n\n"); return;

}

temp = head;

while (temp != NULL) { printf("%d ", temp->data); temp = temp->next;

}

printf("\n");

}

// Function to delete a node at the beginning void deleteAtBeginning() {

if(head == NULL) {

printf("List does not exist\n"); return;

}

if(head == tail) { free(head);

head = tail = NULL;

} else {

temp = head;

head = head->next; free(temp);

}

len--;

printf("\nNode Deleted\n");

}

// Function to delete a node at the end void deleteAtEnd() {

if(head == NULL) {

printf("List does not exist\n"); return;

}

if(tail == head) { free(head);

head = tail = NULL;

} else {

temp = head;

while(temp->next != tail) { temp = temp->next;

}

temp->next = NULL; free(tail);

tail = temp;

}

len--;

printf("\nNode Deleted\n");

}

// Function to delete a node at a specific position void deleteAtPos() {

int pos;

printf("Enter position: "); scanf("%d", &pos);

if(pos > len || pos < 1) { printf("Wrong position entered.\n"); return;

}

if(pos == 1) { deleteAtBeginning(); return;

}

if(pos == len) { deleteAtEnd(); return;

}

temp = head;

while (--pos != 1) { temp = temp->next;

}

struct node \*del = temp->next; temp->next = temp->next->next; free(del);

len--;

printf("\nNode Deleted\n");

}

void main() { int c;

do {

printf("MENU\n1) Insert At End \n2) Insert At Beginning \n3) Insert At Position \n4) Display\n5) Delete Head\n6) Delete Tail\n7) Delete At Pos\nEnter your Choice: ");

scanf("%d", &c); switch (c) {

case 1: insert(); break;

case 2: insertAtBeginning(); break; case 3: insertAtPos(); break;

case 4: display(); break;

case 5: deleteAtBeginning(); break; case 6: deleteAtEnd(); break;

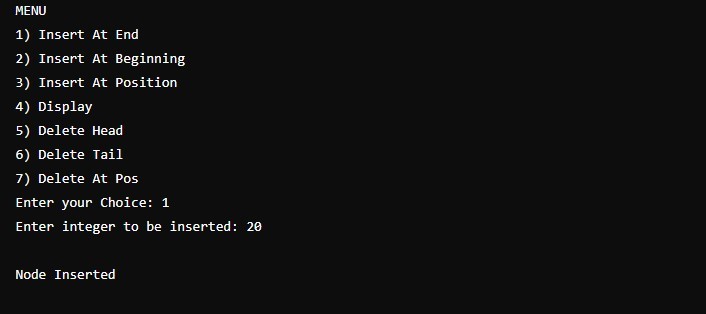
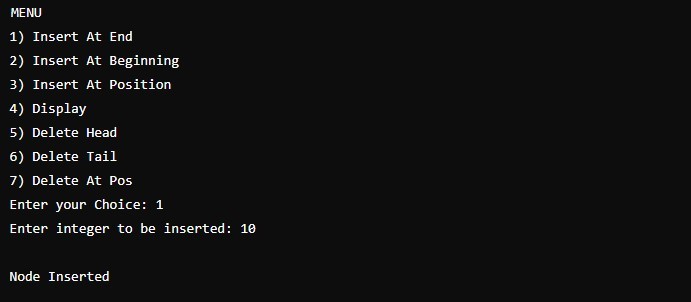
case 7: deleteAtPos(); break; default: break;

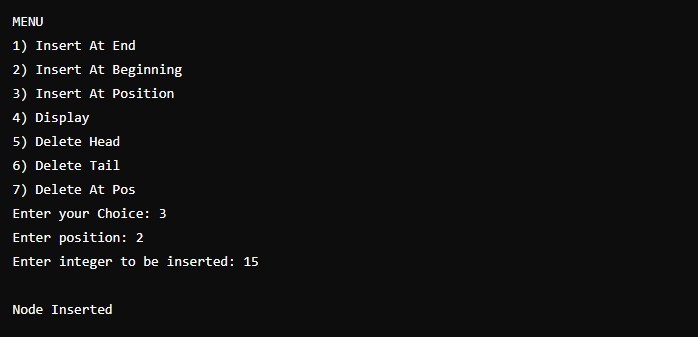
}

} while (c != 0);

printf("\n\n");

}







## PROGRAM-10

### Doubly Linked List

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

// Structure definition for a doubly linked list node struct node {

int data; // Data part of the node struct node \*next; // Pointer to the next node

struct node \*prev; // Pointer to the previous node

};

// Global pointers to keep track of the head and tail of the list struct node \*head = NULL, \*tail = NULL, \*temp;

int len = 0; // Variable to track the length of the list

// Function to insert a node at the end of the list void insert() {

struct node \*newNode = (struct node \*)malloc(sizeof(struct node)); // Allocate memory for a new node

printf("Enter integer to be inserted: "); scanf("%d", &newNode->data);

newNode->next = NULL; // The new node will point to NULL as it will be the last node

if(head == NULL) { // If the list is empty, the new node will be both head and tail

head = newNode; newNode->prev = NULL;

} else { // If the list already exists, link the new node to the tail

tail->next = newNode; newNode->prev = tail;

}

tail = newNode; // Update the tail to the new node len++; // Increment the length of the list printf("\nNode Inserted\n");

}

// Function to insert a node at the beginning of the list void insertAtBeginning() {

struct node \*newNode = (struct node \*)malloc(sizeof(struct node));

printf("Enter integer to be inserted: "); scanf("%d", &newNode->data);

newNode->prev = NULL; // The new node will be the head, so its previous pointer will be NULL

if(head == NULL) { // If the list is empty, the new node will be both head and tail

tail = newNode;

} else { // If the list already exists, update the current head

head->prev = newNode;

}

newNode->next = head; // Point the new node to the current head head = newNode; // Update the head to the new node len++; // Increment the length of the list

printf("\nNode Inserted\n");

}

// Function to insert a node at a specific position void insertAtPos() {

int pos;

printf("Enter position: "); scanf("%d", &pos);

if(pos > len + 1 || pos < 1) { // Validate the position printf("Wrong position entered.\n");

return;

}

if(pos == 1) { // If position is 1, insert at the beginning insertAtBeginning();

return;

}

if(pos == len + 1) { // If position is at the end, use the insert function

insert(); return;

}

struct node \*newNode = (struct node \*)malloc(sizeof(struct node)); printf("Enter integer to be inserted: ");

scanf("%d", &newNode->data);

temp = head;

pos--; // Adjust position for 0-based index while (--pos != 1) {

temp = temp->next; // Traverse to the node before the insertion position

}

newNode->next = temp->next; // Link the new node to the next node temp->next = newNode; // Link the previous node to the new node

newNode->prev = temp; // Set the previous pointer of the new node

newNode->next->prev = newNode; // Update the previous pointer of the next node

len++; // Increment the length of the list printf("\nNode Inserted\n");

}

// Function to display the list from head to tail void display() {

if(head == NULL) { // Check if the list is empty

printf("Empty List\n\n"); return;

}

temp = head;

while (temp != NULL) { // Traverse the list from head to tail printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

displayBack(); // Optionally display the list from tail to head

}

// Function to display the list from tail to head void displayBack() {

if(tail == NULL) { // Check if the list is empty printf("Empty List\n\n");

return;

}

temp = tail;

while (temp != NULL) { // Traverse the list from tail to head printf("%d ", temp->data);

temp = temp->prev;

}

printf("\n");

}

// Function to delete a node at the beginning of the list void deleteAtBeginning() {

if(head == NULL) { // Check if the list exists printf("List does not exist\n"); return;

}

if(head == tail) { // If there's only one node free(head);

head = tail = NULL;

} else {

head = head->next; // Move head to the next node free(head->prev); // Free the old head node

head->prev = NULL; // Set the previous pointer of the new head to

NULL

}

len--; // Decrement the length of the list printf("\nNode Deleted\n");

}

// Function to delete a node at the end of the list void deleteAtEnd() {

if(head == NULL) { // Check if the list exists printf("List does not exist\n"); return;

}

if(tail == head) { // If there's only one node free(head);

head = tail = NULL;

} else {

tail = tail->prev; // Move tail to the previous node free(tail->next); // Free the old tail node

tail->next = NULL; // Set the next pointer of the new tail to

NULL

}

len--; // Decrement the length of the list printf("\nNode Deleted\n");

}

// Function to delete a node at a specific position void deleteAtPos() {

int pos;

printf("Enter position: "); scanf("%d", &pos);

if(pos > len || pos < 1) { // Validate the position printf("Wrong position entered.\n");

return;

}

if(pos == 1) { // If position is 1, delete at the beginning deleteAtBeginning();

return;

}

if(pos == len) { // If position is at the end, delete at the end deleteAtEnd();

return;

}

temp = head;

pos--; // Adjust position for 0-based index

while (--pos != 1) temp = temp->next; // Traverse to the node before the deletion position

struct node \*del = temp->next; // Node to be deleted

temp->next = temp->next->next; // Link previous node to the next node temp->next->prev = temp; // Update the previous pointer of the

next node

free(del); // Free the memory of the deleted node

len--; // Decrement the length of the list printf("\nNode Deleted\n");

}

void main() { int c;

do {

printf("MENU\n1) Insert At End \n2) Insert At Beginning \n3) Insert At Position \n4) Display\n5) Delete Head\n6) Delete Tail\n7) Delete At Pos\nEnter your Choice: ");

scanf("%d", &c); switch (c) {

case 1: insert(); break; // Insert at the end

case 2: insertAtBeginning(); break; // Insert at the

beginning position

position

}

case 3: insertAtPos(); break; // Insert at a specific case 4: display(); break; // Display the list

case 5: deleteAtBeginning(); break; // Delete the head case 6: deleteAtEnd(); break; // Delete the tail

case 7: deleteAtPos(); break; // Delete at a specific default: break;

} while (c != 0); // Repeat until the user enters 0 to exit printf("\n\n");

}

