Practical 1

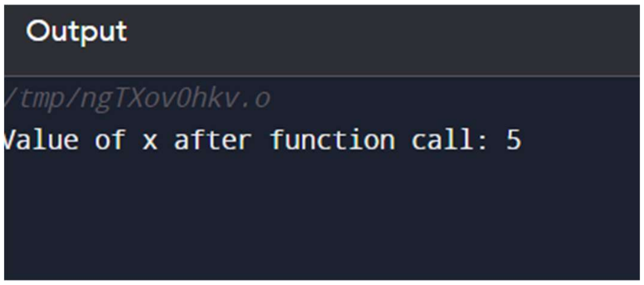
Aim : Introduction to pointers. Call by Value and Call by reference.

Pointers are variables that store the memory address of another variable. They play a crucial role in dynamic memory allocation and are fundamental to many programming concepts. There are two main aspects related to pointers: understanding what pointers are and how they work, and understanding the concepts of call by value and call by reference.

Call by Value:

In a call by value mechanism, the value of the actual parameter is passed to the formal parameter of the function. Changes made to the formal parameter inside the function do not affect the actual parameter.

```
#include <stdio.h>

void square(int num) {

   num = num * num;

}

int main() {

   int x = 5;

   square(x);

   printf("Value of x after function call: %d\n", x); // Output: 5

   return 0;

}
```

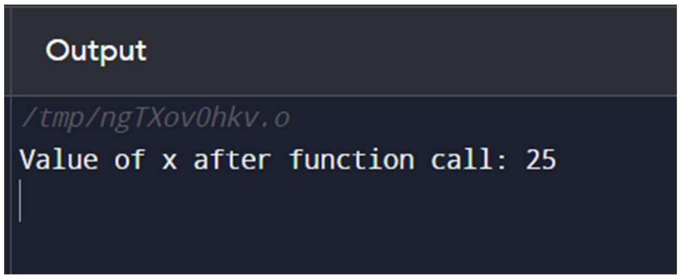

Output

```
/tmp/ngTXov0hkv.o
Value of x after function call: 5
```

Call by Reference:

In a call by reference mechanism, the address of the actual parameter is passed to the formal parameter. Any changes made to the formal parameter inside the function directly affect the actual parameter.

```
#include <stdio.h>

void square(int *num) {

   *num = (*num) * (*num);

}

int main() {

   int x = 5;

   square(&x);

   printf("Value of x after function call: %d\n", x); // Output: 25

   return 0;

}
```



```
/tmp/ngTXov0hkv.o
Value of x after function call: 25
```

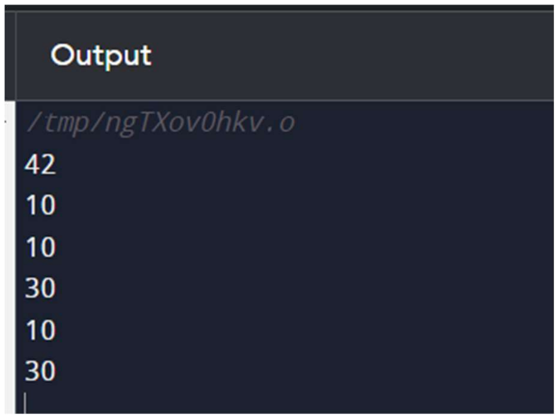
Practical 2

Aim : Introduction to Dynamic Memory Allocation. DMA functions malloc(), calloc(), free() etc.

Dynamic Memory Allocation (DMA) is a process in programming languages where memory is allocated and deallocated during the runtime of a program. It allows for the creation of data structures whose size can change during program execution. In languages like C, dynamic memory allocation is achieved through functions like **malloc()**, **calloc()**, **realloc()**, and **free()**.

Code :

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

int main() {
    // Allocate memory for an integer using malloc
    int *ptr1 = (int *)malloc(sizeof(int));
    if (ptr1 == NULL) {
        printf("Memory allocation using malloc failed.\n");
        return 1;
    }

    *ptr1 = 42;
    printf("%d \n", *ptr1);

    // Allocate memory for an integer array using calloc
    int *ptr2 = (int *)calloc(5, sizeof(int));
    if (ptr2 == NULL) {
        printf("Memory allocation using calloc failed.\n");
        return 1;
    }

     ptr2[0] = 10;
     ptr2[1] = 10;
     ptr2[2] = 30;
     ptr2[3] = 10;
     ptr2[4] = 30;
    for (int i = 0; i < 5; i++) {
       printf("%d \n", ptr2[i]);
    }

    // Deallocate memory
    free(ptr1);
    free(ptr2);

    return 0;
}
```



```
/tmp/ngTXov0hkv.o
42
10
10
30
10
30
```

Practical 3

Aim : Implement a program for stack that performs following operations using array. (a) PUSH (b) POP (c) PEEP (d) CHANGE (e) DISPLAY

Code :

```
#include <stdio.h>
int n =4;

int top = -1, stack[4];
void push();
void pop();
void Display();
void PEEP();

int main(){
    int choice;

    while (1)
    {
        printf("**********************************\n");
        printf("\nPerform operations on the stack:");
        printf("\n1.Push the element\n2.Pop the 
element\n3.Display\n4.PEEP\n5.End");
        printf("\n\nEnter the choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            push();
            break;
        case 2:
            pop();
            break;
        case 3:
            Display();
            break;
        case 4:
            PEEP();
            break;
        case 5:
            return 0;

        default:
            printf("\nInvalid choice!!");
        }
    }
}
```

```c
void push()
{
    int x;

    if (top == n - 1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter the element : ");
        scanf("%d", &x);
        top = top + 1;
        stack[top] = x;
    }
}
void pop()
{
    if (top == -1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element: %d", stack[top]);
        top = top - 1;
    }
}
void PEEP(){

    if (top == -1)
    {
        printf("\nUnderflow!!");
    } else{
        printf("Last Element :%d",stack[top]);
    }
}
void Display()
{
    if (top == -1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
    //     for (int i = top; i >= 0; --i)
    //         printf("%d\n", stack[i]);
    // }
    for (int i = 0; i <= top; i++)
    {
        printf("%d\n", stack[i]);
    }
}

}
```

Output :

```
**********************************

Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 1


Enter the element : 5

**********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 1


Enter the element : 2
```

```
*********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 1


Enter the element : 6

*********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 3


Elements present in the stack:

5

2

6

*********************************


Perform operations on the stack:

1.Push the element
```

```
2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 2


Popped element: 6**********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 3


Elements present in the stack:

5

2

**********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 4
```

```
Last Element :2**********************************


Perform operations on the stack:

1.Push the element

2.Pop the element

3.Display

4.PEEP

5.End


Enter the choice: 5
```

Practical 4

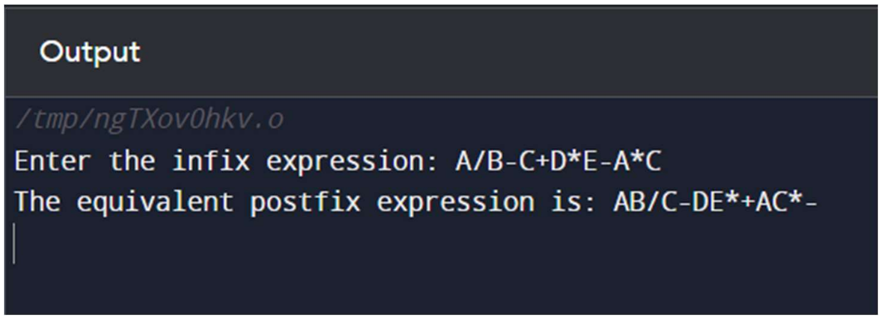
Aim : Implement a program to convert infix notation to postfix notation using stack.

Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100

char stack[MAX];
char infix[MAX], postfix[MAX];
int top = -1;

void push(char);
char pop();
int isEmpty();
void inToPost();
void print();
int precedence(char);

int main()
{
    printf("Enter the infix expression: ");
    gets(infix);
    inToPost();
    print();
    return 0;
}

void inToPost()
{
    int i, j = 0;
    char symbol, next;
    for (i = 0; i < strlen(infix); i++)
    {
        symbol = infix[i];
        switch (symbol)
        {
        case '(':
            push(symbol);
            break;
        case ')':
            while ((next = pop()) != '(')
                postfix[j++] = next;
            break;
        case '+':
        case '-':
        case '*':
```

```
        case '/':
        case '^':
            while (!isEmpty() && precedence(stack[top]) >= precedence(symbol))
                postfix[j++] = pop();
            push(symbol);
            break;
        default:
            postfix[j++] = symbol;
        }
    }

    while (!isEmpty())
        postfix[j++] = pop();
    postfix[j] = '\0';
}

int precedence(char symbol)
{
    switch (symbol)
    {
    // Higher value means higher precedence
    case '^':
        return 3;
    case '/':
    case '*':
        return 2;
    case '+':
    case '-':
        return 1;
    default:
        return 0;
    }
}

void print()
{
    int i = 0;
    printf("The equivalent postfix expression is: ");
    while (postfix[i])
    {
        printf("%c", postfix[i++]);
    }
    printf("\n");
}
```

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

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

int isEmpty()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
```



Output

```
/tmp/ngTXov0hkv.o
Enter the infix expression: A/B-C+D*E-A*C
The equivalent postfix expression is: AB/C-DE*+AC*-
```

Practical 5

Aim : Write a program to implement QUEUE using arrays that performs following operations (a) INSERT (b) DELETE (c) DISPLAY

Code :

```c
#include <stdio.h>
// #include <stdlib.h>
// #define n 10

void enqueue();
void dequeue();
void show();

int n = 5;
int queue[5];
int Rear = -1;
int Front = -1;

int main()
{
    int ch;
    while (1)
    {
        printf("\n-------------------------------------\n");
        printf("1.Enqueue\n");
        printf("2.Dequeue\n");
        printf("3.Display\n");
        printf("4.Exit\n");
        printf("Enter operations : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            enqueue();
            break;
        case 2:
            dequeue();
            break;
        case 3:
            show();
            break;
        case 4:
            return 0;
            break;
        default:
            printf("Incorrect choice \n");
        }
    }
```

```c
}

void enqueue()
{
    int data;
    if (Rear == n - 1)
        printf("Overflow \n");
    else
    {
        if (Front == -1)
        {
            Front = 0;
        }

        printf("Enter Element \n : ");
        scanf("%d", &data);
        Rear = Rear + 1;
        queue[Rear] = data;
    }
}

void dequeue()
{
    if (Front == -1 && Rear == -1)
    {
        printf("Underflow \n");
    }
    else if (Front == Rear)
    {
        Front = Rear = -1;
    }
    else
    {
        printf("Element deleted : %d\n", queue[Front]);
        Front = Front + 1;
    }
}

void show()
{

    if (Front == -1)
    {
        printf("Empty Queue \n");
    }
    else
    {
        printf("\n************************\n");
        printf("Queue:");
        for (int i = Front; i <= Rear; i++)
        {
            printf("%d ", queue[i]);
        }
        printf("\n*************************\n");
    }
}
```

Practical 6

Aim : Write a program to implement Circular Queue using arrays that performs following operations. (a) INSERT (b) DELETE (c) DISPLAY

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

#define MAX_SIZE 5

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

// Function to check if the queue is empty
int isEmpty() {
    return (front == -1);
}

// Function to check if the queue is full
int isFull() {
    return ((front == 0 && rear == MAX_SIZE - 1) || (rear == (front - 1) %
(MAX_SIZE - 1)));
}

// Function to enqueue an element to the circular queue
void enqueue(int data) {
    if (isFull()) {
        printf("Queue is full. Cannot enqueue %d\n", data);
        return;
    }

    if (isEmpty())
        front = 0;

    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = data;
    printf("%d enqueued to the queue\n", data);
}

// Function to dequeue an element from the circular queue
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue\n");
        return -1;
    }

    int data = queue[front];
    if (front == rear)
        front = rear = -1; // Queue is now empty
    else
        front = (front + 1) % MAX_SIZE;

    printf("%d dequeued from the queue\n", data);
    return data;
}

// Function to display the elements in the circular queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }

    printf("Elements in the queue: ");
    int i = front;
    do {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (rear + 1) % MAX_SIZE);

    printf("\n");
}

int main() {
    int choice, data;

    while (1) {
        printf("\nCircular Queue Menu:\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 data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Practical 7

Aim : Write a menu driven program to implement following operations on the singly linked list. (a) Insert a node at the front of the linked list. (b) Insert a node at the end of the linked list. (c) Insert a node such that linked list is in ascending order.(according to info. Field) (d) Delete a first node of the linked list. (e) Delete a node before specified position. (f) Delete a node after specified position.

Code :

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

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

struct Node* head = NULL;

// Function to display the linked list
void displayList() {
    struct Node* temp = head;
    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to insert a node at the front of the linked list
void insertAtFront(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
      if (head == NULL) {
        head = newNode;
      } else{
    newNode->next = head;
    head = newNode;
      }
    printf("%d inserted at the front.\n", value);
}

// Function to insert a node at the end of the linked list
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    printf("%d inserted at the end.\n", value);
}

// Function to insert a node in ascending order
void insertInAscending(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (head == NULL || value < head->data) {
        newNode->next = head;
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL && temp->next->data < value) {
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }

    printf("%d inserted in ascending order.\n", value);
}

// Function to delete the first node of the linked list
void deleteFirstNode() {
    if (head == NULL) {
        printf("Linked List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    head = head->next;
    printf("%d deleted from the front.\n", temp->data);
    free(temp);
}

// Function to delete a node before a specified position
```

```c
void deleteBeforePosition(int position) {
    if (head == NULL || position <= 1) {
        printf("Invalid position or Linked List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

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

    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position. Cannot delete.\n");
        return;
    }

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

    printf("Node before position %d deleted.\n", position);
    free(temp);
}

// Function to delete a node after a specified position
void deleteAfterPosition(int position) {
    if (head == NULL || position < 0) {
        printf("Invalid position or Linked List is empty. Cannot delete.\n");
        return;
    }

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

    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position. Cannot delete.\n");
        return;
    }

    struct Node* deleteNode = temp->next;
    temp->next = deleteNode->next;
    printf("Node after position %d deleted.\n", position);
    free(deleteNode);
}

int main() {
    int choice, value, position;

    while (1) {
        printf("\nMENU\n");
        printf("1. Insert at Front\n");
        printf("2. Insert at End\n");
        printf("3. Insert in Ascending Order\n");
        printf("4. Delete First Node\n");
        printf("5. Delete Before Position\n");
        printf("6. Delete After Position\n");
        printf("7. Display\n");
        printf("8. Exit\n");

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

        switch (choice) {
            case 1:
                printf("Enter the value to insert at the front: ");
                scanf("%d", &value);
                insertAtFront(value);
                break;

            case 2:
                printf("Enter the value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;

            case 3:
                printf("Enter the value to insert in ascending order: ");
                scanf("%d", &value);
                insertInAscending(value);
                break;

            case 4:
                deleteFirstNode();
                break;

            case 5:
                printf("Enter the position before which node should be
deleted: ");
```

```
                scanf("%d", &position);
                deleteBeforePosition(position);
                break;

            case 6:
                printf("Enter the position after which node should be deleted: 
");
                scanf("%d", &position);
                deleteAfterPosition(position);
                break;

            case 7:
                displayList();
                break;

            case 8:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Practical 8

Aim : Write a program to implement stack using linked list.

Code :

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

struct Node {
    int data;
    struct Node* next;
};
struct Node* top = NULL;

// Function to check if the stack is empty
int isEmpty() {
    return top == NULL;
}
// Function to push an element onto the stack
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed. Stack overflow.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed onto the stack.\n", value);
}
// Function to pop an element from the stack
void pop() {
    if (isEmpty()) {
        printf("Stack is empty. Cannot pop.\n");
        return;
    }
    struct Node* temp = top;
    top = top->next;
    printf("%d popped from the stack.\n", temp->data);
    free(temp);
}
// Function to display the elements in the stack
void display() {
    if (isEmpty()) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
```

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

int main() {
    int choice, value;

    while (1) {
        printf("\nMENU\n");
        printf("1. Push\n");
        printf("2. Pop\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 push onto the stack: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }
    return 0;
}
```

Practical 9

Aim : Write a program to implement queue using linked list

Code :

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

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

struct Node* front = NULL;
struct Node* rear = NULL;

// Function to check if the queue is empty
int isEmpty() {
    return front == NULL;
}

// Function to enqueue an element into the queue
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed. Queue overflow.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;

    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }

    printf("%d enqueued into the queue.\n", value);
}

// Function to dequeue an element from the queue
void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue.\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    // If the queue becomes empty after dequeue
    if (front == NULL) {
        rear = NULL;
    }

    printf("%d dequeued from the queue.\n", temp->data);
    free(temp);
}

// Function to display the elements in the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\nMENU\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 the value to enqueue into the queue: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Practical 10

Aim : Write a program to implement following operations on the doubly linked list. (a) Insert a node at the front of the linked list. (b) Insert a node at the end of the linked list. (c) Delete a last node of the linked list. (d) Delete a node before specified position.

Code :

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

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

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");
        exit(1);
    }
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to display the doubly linked list
void displayList() {
    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to insert a node at the front of the linked list
void insertAtFront(int value) {
    struct Node* newNode = createNode(value);

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

    printf("%d inserted at the front.\n", value);
}

// Function to insert a node at the end of the linked list
void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }

    printf("%d inserted at the end.\n", value);
}

// Function to delete the last node of the linked list
void deleteLastNode() {
    if (head == NULL) {
        printf("Linked List is empty. Cannot delete.\n");
        return;
    }

    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Last node deleted.\n");
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->prev->next = NULL;
    free(temp);
    printf("Last node deleted.\n");
```

```c
}

// Function to delete a node before a specified position
void deleteBeforePosition(int position) {
    if (head == NULL || position <= 1) {
        printf("Invalid position or Linked List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;

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

    if (temp == NULL || temp->prev == NULL) {
        printf("Invalid position. Cannot delete.\n");
        return;
    }

    struct Node* deleteNode = temp->prev;
    temp->prev = deleteNode->prev;

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

    free(deleteNode);
    printf("Node before position %d deleted.\n", position);
}

int main() {
    int choice, value, position;

    while (1) {
        printf("\nMENU\n");
        printf("1. Insert at Front\n");
        printf("2. Insert at End\n");
        printf("3. Delete Last Node\n");
        printf("4. Delete Before Position\n");
        printf("5. Display\n");
        printf("6. Exit\n");

        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the value to insert at the front: ");
                scanf("%d", &value);
                insertAtFront(value);
                break;

            case 2:
                printf("Enter the value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;

            case 3:
                deleteLastNode();
                break;

            case 4:
                printf("Enter the position before which node should be 
deleted: ");
                scanf("%d", &position);
                deleteBeforePosition(position);
                break;

            case 5:
                displayList();
                break;

            case 6:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Practical 11

Aim : Write a program to implement following operations on the circular linked list. (a) Insert a node at the end of the linked list. (b) Insert a node before specified position. (c) Delete a first node of the linked list. (d) Delete a node after specified position.

Code :

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

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

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");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to display the circular linked list
void displayList() {
    if (head == NULL) {
        printf("Circular Linked List is empty.\n");
        return;
    }

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

    printf("(Head)\n");
}

// Function to insert a node at the end of the circular linked list
void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }

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

    printf("%d inserted at the end.\n", value);
}

// Function to insert a node before a specified position in the circular
linked list
void insertBeforePosition(int position, int value) {
    if (position <= 0) {
        printf("Invalid position. Cannot insert.\n");
        return;
    }

    struct Node* newNode = createNode(value);

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
        printf("%d inserted before position %d.\n", value, position);
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;
    int count = 0;

    do {
        prev = temp;
        temp = temp->next;
        count++;
    } while (temp != head && count < position - 1);

    if (count == 0) {
        newNode->next = head;
        head = newNode;
        struct Node* last = head;
        while (last->next != head) {
            last = last->next;
        }
        last->next = head;
    } else if (count == position - 1) {
        prev->next = newNode;
        newNode->next = temp;
    } else {
        printf("Invalid position. Cannot insert.\n");
        free(newNode);
        return;
    }

    printf("%d inserted before position %d.\n", value, position);
}

// Function to delete the first node of the circular linked list
void deleteFirstNode() {
    if (head == NULL) {
        printf("Circular Linked List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;

    if (head->next == head) {
        head = NULL;
    } else {
        while (temp->next != head) {
            temp = temp->next;
        }

        temp->next = head->next;
        head = head->next;
    }

    free(temp);
    printf("First node deleted.\n");
}

// Function to delete a node after a specified position in the circular linked
list
void deleteAfterPosition(int position) {
    if (position < 0) {
        printf("Invalid position. Cannot delete.\n");
        return;
    }

    if (head == NULL) {
        printf("Circular Linked List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;
    int count = 0;

    do {
        prev = temp;
        temp = temp->next;
        count++;
    } while (temp != head && count < position);

    if (count == position) {
        if (temp->next == head) {
            printf("No node after position %d. Cannot delete.\n", position);
            return;
        }

        prev = temp;
        temp = temp->next;
        prev->next = temp->next;
        free(temp);
        printf("Node after position %d deleted.\n", position);
    } else {
        printf("Invalid position. Cannot delete.\n");
    }
}

int main() {
    int choice, position, value;

    while (1) {
        printf("\nMENU\n");
        printf("1. Insert at End\n");
        printf("2. Insert Before Position\n");
        printf("3. Delete First Node\n");
        printf("4. Delete After Position\n");
        printf("5. Display\n");
        printf("6. Exit\n");

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

        switch (choice) {
            case 1:
                printf("Enter the value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;

            case 2:
                printf("Enter the position before which node should be 
inserted: ");
                scanf("%d", &position);
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                insertBeforePosition(position, value);
                break;

            case 3:
                deleteFirstNode();
                break;

            case 4:
                printf("Enter the position after which node should be deleted: 
");
                scanf("%d", &position);
                deleteAfterPosition(position);
                break;

            case 5:
                displayList();
                break;

            case 6:
                exit(0);

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Practical 12

Aim : Write a program which create binary search tree.

Code :

```c
// Binary Search Tree operations in C

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

struct node {
  int key;
  struct node *left;
  struct node *right;
};

// Create a node
struct node *newNode(int data) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = data;
  temp->left = NULL;
  temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    inorder(root->left);
    printf("%d -> ", root->key);
    inorder(root->right);
  }
}

// Insert a node
struct node *insert(struct node *root, int key) {
  // Return a new node if the tree is empty
  if (root == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key <= root->key)
    root->left = insert(root->left, key);
  else
    root->right = insert(root->right, key);

  return root;
}

int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  insert(root, 3);
  insert(root, 1);
  insert(root, 6);
  insert(root, 7);
  insert(root, 10);
  insert(root, 14);
  insert(root, 4);

  printf("Inorder traversal: ");
  inorder(root);

}
```

Practical 13

Aim : Implement recursive and non-recursive tree traversing methods inorder, preorder and postorder traversal

Code :

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

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

// Function to create a new node
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 BST
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;
}

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

```c
// Recursive Preorder Traversal
void preorderRecursive(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderRecursive(root->left);
        preorderRecursive(root->right);
    }
}

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

// Non-recursive Inorder Traversal using a Stack
void inorderNonRecursive(struct Node* root) {
    struct Node* stack[100];
    int top = -1;

    while (root != NULL || top != -1) {
        while (root != NULL) {
            stack[++top] = root;
            root = root->left;
        }

        root = stack[top--];
        printf("%d ", root->data);

        root = root->right;
    }
}

// Non-recursive Preorder Traversal using a Stack
void preorderNonRecursive(struct Node* root) {
    struct Node* stack[100];
    int top = -1;

    while (root != NULL || top != -1) {
        while (root != NULL) {
            printf("%d ", root->data);
            stack[++top] = root;
            root = root->left;
        }
        root = stack[top--];
        root = root->right;
    }
}

// Non-recursive Postorder Traversal using two Stacks
void postorderNonRecursive(struct Node* root) {
    struct Node* stack1[100];
    struct Node* stack2[100];
    int top1 = -1, top2 = -1;

    stack1[++top1] = root;

    while (top1 != -1) {
        root = stack1[top1--];
        stack2[++top2] = root;

        if (root->left != NULL) {
            stack1[++top1] = root->left;
        }

        if (root->right != NULL) {
            stack1[++top1] = root->right;
        }
    }

    while (top2 != -1) {
        printf("%d ", stack2[top2--]->data);
    }
}

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

    root = insert(root, 8);
    insert(root, 3);
    insert(root, 1);
    insert(root, 6);
    insert(root, 7);
    insert(root, 10);
    insert(root, 14);
    insert(root, 4);

    printf("Recursive Inorder Traversal: ");
    inorderRecursive(root);
    printf("\n");

    printf("Recursive Preorder Traversal: ");
    preorderRecursive(root);
    printf("\n");

    printf("Recursive Postorder Traversal: ");
    postorderRecursive(root);
    printf("\n");

    printf("Non-recursive Inorder Traversal: ");
    inorderNonRecursive(root);
    printf("\n");

    printf("Non-recursive Preorder Traversal: ");
    preorderNonRecursive(root);
    printf("\n");

    printf("Non-recursive Postorder Traversal: ");
    postorderNonRecursive(root);
    printf("\n");

    return 0;
}
```

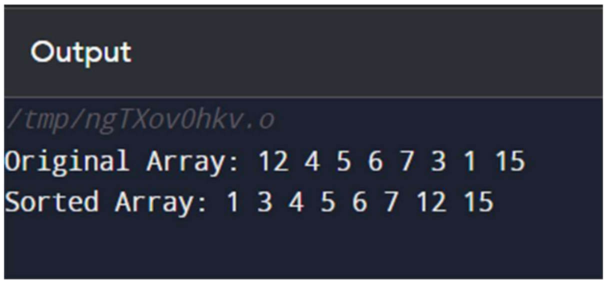
Practical 14

Aim : Write a program to implement Quick Sort

Code :

```c
#include <stdio.h>

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Swap arr[i+1] and arr[high] (pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// Function to perform Quick Sort on the array
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Find the pivot index such that elements smaller than pivot are on
the left, and greater are on the right
        int pivotIndex = partition(arr, low, high);

        // Recursively sort the subarrays
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```c
int main() {
    int arr[] = {12, 4, 5, 6, 7, 3, 1, 15};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted Array: ");
    printArray(arr, n);

    return 0;
}
```



```
/tmp/ngTXovOhkv.o
Original Array: 12 4 5 6 7 3 1 15
Sorted Array: 1 3 4 5 6 7 12 15
```

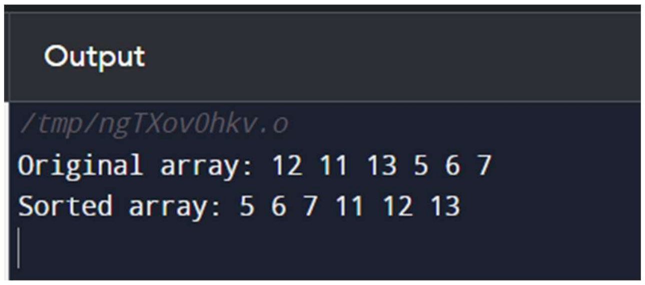
Practical 15

Aim : Write a program to implement Merge Sort

Code :

```c
#include <stdio.h>

void merge(int arr[], int left[], int right[], int left_size, int right_size)
{
    int i = 0, j = 0, k = 0;

    while (i < left_size && j < right_size)
    {
        if (left[i] <= right[j])
        {
            arr[k] = left[i];
            k++;
            i++;
        }
        else
        {
            arr[k] = right[j];
            k++;
            j++;
        }
    }

    while (i < left_size)
    {
        arr[k] = left[i];
        k++;
        i++;
    }

    while (j < right_size)
    {
        arr[k] = right[j];
        k++;
        j++;
    }
}

void mergeSort(int arr[], int size)
{
    int i;
    if (size <= 1)
    {
        return; // Base case: already sorted
    }

    int mid = size / 2;
    int left[mid];
    int right[size - mid];

    // Split the array into two halves
    for (i = 0; i < mid; i++)
    {
        left[i] = arr[i];
    }

    for (i = mid; i < size; i++)
    {
        right[i - mid] = arr[i];
    }

    // Recursively sort the left and right halves
    mergeSort(left, mid);
    mergeSort(right, size - mid);

    // Merge the sorted halves
    merge(arr, left, right, mid, size - mid);
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = 6;

    printf("Original array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");

    mergeSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
```

```
}
```

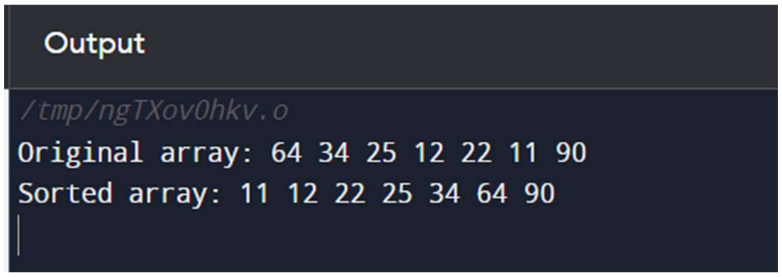


```
/tmp/ngTXov0hkv.o
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

Practical 16

Aim : Write a program to implement Bubble Sort

```c
#include <stdio.h>
void printArray(int arr[], int n);
void bubbleSort(int arr[], int n);

void bubbleSort(int arr[], int n) {
    int temp;
    int swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }

        // If no two elements were swapped in the inner loop, the array is
already sorted
        if (swapped == 0) {
            break;
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr) / sizeof(arr[0]);
    int n=7;

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr,n);

    return 0;
}

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



Output

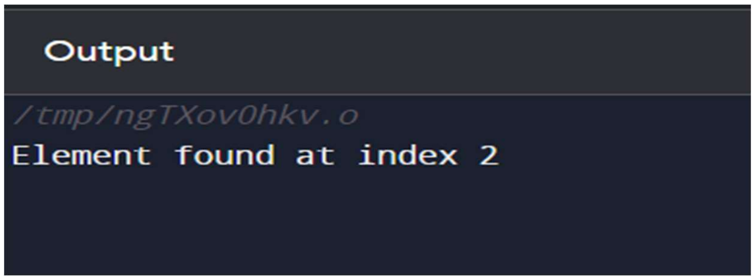
```
/tmp/ngTXov0hkv.o
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

Practical 17

Aim : . Write a program to implement Binary Search.

Code :

```c
#include <stdio.h>
int binarySearch(int arr[], int n, int key) {
    int s = 0;
    int e = n - 1;
    int mid;
    while (s <= e) {
         mid = (s+e)/2;

        if (arr[mid] == key) {
            return mid; // key found at index 'mid'
        } else if(arr[mid] < key) {
            s = mid + 1; // Adjust s boundary
        } else {
            e = mid - 1; // Adjust e boundary
        }
    }
    return -1; // key not found
}
int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 45, 56, 72};
    // int n = nof(arr) / nof(arr[0]);
    int n = 10;
    int key = 8;

    int result = binarySearch(arr, n, key);

    if (result == -1) {
        printf("Element not found in the array\n");
    } else {
        printf("Element found at index %d\n",result);
    }

    return 0;
}
```



Output

```
/tmp/ngTXov0hkv.o
Element found at index 2
```