Experiment - 1

a) Write a program in C to display the n terms of even natural number and their sum.

Source Code:

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("The even numbers are: ");
    for (i = 1; i <= n; i++)
    {
        int even = 2 * i;
        printf("%d ", even);
        sum += even;
    }
    printf("\nThe sum of the even numbers is: %d\n", sum);
    return 0;
}
```

OUTPUT:

b) Write a program in C to display the n terms of harmonic series and their sum. 1 + 1/2 + 1/3 + ¼ + 1/5 ... 1/n terms.

Source Code:

```
#include <stdio.h>
int main()
{
    int n, i;
    float s = 0.0;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("The harmonic series is: ");
    for (i = 1; i <= n; i++)
    {
        float term = 1.0 / i;
        printf("%.2f ", term);
        s += term;
    }
    printf("\nThe sum of the harmonic series is: %.2f\n", s);
    return 0;
}
```

OUTPUT:

c) Write a C program to check whether a given number is an Armstrong number or not.

Source Code:

```
#include <stdio.h>
#include <math.h>
int main()
{
    int num, originalNum, remainder, n = 0;
    float result = 0.0;
    printf("Enter an integer: ");
    scanf("%d", &num);
    originalNum = num;
    for (originalNum = num; originalNum != 0; ++n) {
        originalNum /= 10; }
    originalNum = num;
    while (originalNum != 0) {
        remainder = originalNum % 10;
        result += pow(remainder, n);
        originalNum /= 10; }
    if ((int)result == num)
        printf("%d is an Armstrong number.", num);
    else
        printf("%d is not an Armstrong number.", num);
}
```

OUTPUT:

d) Write a C program to calculate the factorial of a given number.

Source Code:

```
#include <stdio.h>
int main()
{
   int n, i;
   unsigned long long fact = 1;
   printf("Enter an integer: ");
   scanf("%d", &n);
   for (i = 1; i <= n; i++)
   {
      fact = fact * i;
   }
   printf("Factorial of %d = %llu\n", n, fact);
   return 0;
}
```

OUTPUT:

Experiment-2:

a) Write a program in C for multiplication of two square Matrices.

Source Code:

```
#include <stdio.h>
#define SIZE 3
int main()
{
    int A[SIZE][SIZE];
    int B[SIZE][SIZE];
    int C[SIZE][SIZE];
    int row, col, i, sum;
    printf("Enter elements in matrix A of size %dx%d: \n", SIZE, SIZE);
    for(row=0; row<SIZE; row++)
    {
        for(col=0; col<SIZE; col++)
        {
            scanf("%d", &A[row][col]);
        }
    }
    printf("\nEnter elements in matrix B of size %dx%d: \n", SIZE, SIZE);
    for(row=0; row<SIZE; row++)
    {
        for(col=0; col<SIZE; col++)
        {
            scanf("%d", &B[row][col]);
        }
    }
    for(row=0; row<SIZE; row++)
    {
        for(col=0; col<SIZE; col++)
        {
            sum = 0;
            for(i=0; i<SIZE; i++)
            {
                sum += A[row][i] * B[i][col];
            }
            C[row][col] = sum;
        }
    }
    printf("\nProduct of matrix A * B = \n");
    for(row=0; row<SIZE; row++)
    {
        for(col=0; col<SIZE; col++)
        {
            printf("%d ", C[row][col]);
        }
```

```
        printf("\n");
    }
    return 0;
}
```

OUTPUT:

b) Write a program in C to find transpose of a given matrix.

Source Code:

```
#include <stdio.h>
int main()
{
    int n; a
    printf("Enter the size of the matrices: ");
    scanf("%d", &n);
    int A[n][n];
    int B[n][n];
    int C[n][n];
    int i, j, k, sum;
    printf("Enter the elements of the first matrix: \n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("Enter A[%d][%d]: ", i, j);
            scanf("%d", &A[i][j]);
        }
    }
    printf("Enter the elements of the second matrix: \n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("Enter B[%d][%d]: ", i, j);
            scanf("%d", &B[i][j]);
        }
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            sum = 0;
            for (k = 0; k < n; k++)
            {  sum += A[i][k] * B[k][j];  }
            C[i][j] = sum;
        }
    }
    printf("The product of the two matrices is: \n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%d ", C[i][j]);
```

```
        }
        printf("\n");
    }
    return 0;
}
```

OUTPUT:

Experiment-3

a) Write a program in C to check whether a number is a prime number or not using the function.

Source Code:

```
#include <stdio.h>
#include <math.h>
int isPrime(int n)
{
    if (n <= 1)
        return 0;
    if (n <= 3)
        return 1;
    if (n % 2 == 0 || n % 3 == 0)
        return 0;
    for (int i = 5; i <= sqrt(n); i += 6)
    {
        if (n % i == 0 || n % (i + 2) == 0)
            return 0;
    }
    return 1;
}

int main()
{
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (isPrime(n))
        printf("%d is a prime number.\n", n);
    else
        printf("%d is not a prime number.\n", n);
    return 0;
}
```

OUTPUT:

b) Write recursive program which computes the nth Fibonacci number, for appropriate values of n.

Source Code:

```
#include <stdio.h>
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return n;
  }
  else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}

int main() {
   printf("%d\n", fibonacci(10));
  return 0;
}
```

OUTPUT:

c) Write a program in C to add numbers using call by reference.

Source Code:

```
#include <stdio.h>
long addTwoNumbers(long *, long *);
int main()
{
   long num1, num2, sum;
   printf("Enter two numbers to add: ");
   scanf("%ld %ld", &num1, &num2);
   sum = addTwoNumbers(&num1, &num2);
   printf("The sum of %ld and %ld is %ld\n", num1, num2, sum);
   return 0;
}
long addTwoNumbers(long *n1, long *n2)
{
   long sum;
   sum = *n1 + *n2;
   return sum;
}
```

OUTPUT:

Experiment-4:

a) Write a program in C to append multiple lines at the end of a text file.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *file;
    char lines[][100] = {"Line 1: Appended line 1", "Line 2: Appended line 2", "Line 3: Appended line
3"};
    int num_lines = sizeof(lines) / sizeof(lines[0]);
    int i;
    file = fopen("example.txt", "a");
    if (file == NULL) {
        printf("Error opening the file.\n");
        return 1;
    }
            for (i = 0; i < num_lines; i++) {
        fprintf(file, "%s\n", lines[i]);
    }
    fclose(file);
    printf("Lines appended successfully.\n");
    return 0;
}
```

OUTPUT:

b) Write a program in C to copy a file in another name.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *source_file, *destination_file;
   char source_filename[] = "source.txt";
   char destination_filename[] = "destination.txt";
   char ch;
   source_file = fopen(source_filename, "r");
   if (source_file == NULL) {
      printf("Error opening the source file.\n");
      return 1;
   }
            destination_file = fopen(destination_filename, "w");
   if (destination_file == NULL) {
      printf("Error opening the destination file.\n");
      fclose(source_file);
      return 1;
   }
   while ((ch = fgetc(source_file)) != EOF) {
      fputc(ch, destination_file);
}
   fclose(source_file);
   fclose(destination_file);
   printf("File copied successfully.\n");
   return 0;
}
```

OUTPUT:

Experiment-5:

Write recursive program for the following

a) Write recursive and non recursive C program for calculation of Factorial of an integer.

Source Code:

```
#include <stdio.h>
int factorial_recursive(int n) {
   if (n == 0 || n == 1) {
      return 1;
   } else {
      return n * factorial_recursive(n - 1);
   }
}
         int factorial_non_recursive(int n) {
   int result = 1;
   while (n > 1) {
      result *= n;
      n--;
   }
   return result;
}
int main() {
   int n;
   printf("Enter a number: ");
   scanf("%d", &n);
   if (n < 0) {
      printf("Factorial is not defined for negative numbers.\n");
   } else {
      int result = factorial_non_recursive(n);
      printf("Non recussion Factorial of %d is %d.\n", n, result);
   return 0;
}
```

OUTPUT:

b) Write recursive and non recursive C program for calculation of GCD (n, m).

Source Code:

```
#include <stdio.h>
// Function to calculate GCD recursively
int gcd_recursive(int n, int m) {
    if (m == 0) {
        return n;
    } else {
        return gcd_recursive(m, n % m);
    }
}
// Function to calculate GCD non-recursively
int gcd_non_recursive(int n, int m) {
    int temp;
    while (m != 0) {
        temp = n;
        n = m;
        m = temp % m;
    }
    return n;
}
int main() {
    int n, m;
    printf("Enter two numbers: ");
    scanf("%d %d", &n, &m);

    if (n < 0 || m < 0) {
        printf("GCD is not defined for negative numbers.\n");
    } else {
        int result_recursive = gcd_recursive(n, m);
        int result_non_recursive = gcd_non_recursive(n, m);

        printf("GCD of %d and %d (recursive) is: %d\n", n, m, result_recursive);
        printf("GCD of %d and %d (non-recursive) is: %d\n", n, m, result_non_recursive);
    }

    return 0;
}
```

OUTPUT:

c) Write recursive and non recursive C program for Towers of Hanoi: N disks are to be transferred from peg S to peg D with Peg I as the intermediate peg.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
void moveDisk(char source, char destination, int disk) {
    printf("Move disk %d from peg %c to peg %c\n", disk, source, destination);
}
void towersOfHanoiRecursive(int disks, char source, char destination, char intermediate) {
    if (disks == 1) {
        moveDisk(source, destination, 1);
        return;
    }
    towersOfHanoiRecursive(disks - 1, source, intermediate, destination);
    moveDisk(source, destination, disks);
    towersOfHanoiRecursive(disks - 1, intermediate, destination, source);
}
void towersOfHanoiNonRecursive(int disks, char source, char destination, char intermediate) {
    int total_moves = (1 << disks) - 1; // Total moves required
    int i;
    char temp;
    if (disks % 2 == 0) {
        temp = destination;
        destination = intermediate;
        intermediate = temp;
    }
    for (i = 1; i <= total_moves; ++i) {
        if (i % 3 == 1)
            moveDisk(source, destination, i);
        else if (i % 3 == 2)
            moveDisk(source, intermediate, i);
        else if (i % 3 == 0)
            moveDisk(intermediate, destination, i);
    }
}
int main() {
    int disks;
    printf("Enter the number of disks: ");
    scanf("%d", &disks);
    if (disks <= 0) {
        printf("Number of disks should be a positive integer.\n");
        return 1;
    }
    printf("Recursive approach:\n");
    towersOfHanoiRecursive(disks, 'S', 'D', 'I');
```

```
    printf("\nNon-recursive approach:\n");
    towersOfHanoiNonRecursive(disks, 'S', 'D', 'I');
    return 0;
}
```

OUTPUT:

Experiment-6:

a) Write C program that use both recursive and non recursive functions to perform Linear search for a Key value in a given list.

Source Code:

```
#include <stdio.h>
int linearSearchNonRecursive(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i; // Key found, return index
    }
    return -1; // Key not found
}
int linearSearchRecursive(int arr[], int key, int index, int n) {
    if (index >= n)
        return -1; // Key not found
    if (arr[index] == key)
        return index; // Key found, return index
    return linearSearchRecursive(arr, key, index + 1, n); // Recursive call
}

int main() {
    int n, key;
    printf("Enter the number of elements in the list: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the key value to search: ");
    scanf("%d", &key);
    int result_non_recursive = linearSearchNonRecursive(arr, n, key);
    if (result_non_recursive != -1) {
        printf("Key %d found at index %d (non-recursive).\n", key, result_non_recursive);
    } else {
        printf("Key %d not found (non-recursive).\n", key);
    }
    int result_recursive = linearSearchRecursive(arr, key, 0, n);
    if (result_recursive != -1) {
        printf("Key %d found at index %d (recursive).\n", key, result_recursive);
    } else {
        printf("Key %d not found (recursive).\n", key);
    }
 return 0;
}
```

OUTPUT:

b) Write C program that use both recursive and non recursive functions to perform Binary search for a Key value in a given list.

Source Code:

```
#include <stdio.h>
int binarySearchNonRecursive(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid; // Key found, return index
        else if (arr[mid] < key)
            low = mid + 1; // Key is in the right half
        else
            high = mid - 1; // Key is in the left half
    }
    return -1; // Key not found
}
int binarySearchRecursive(int arr[], int key, int low, int high) {
    if (low > high)
        return -1; // Key not found
    int mid = low + (high - low) / 2;
    if (arr[mid] == key)
        return mid; // Key found, return index
    else if (arr[mid] < key)
        return binarySearchRecursive(arr, key, mid + 1, high); // Search in the right half
    else
        return binarySearchRecursive(arr, key, low, mid - 1); // Search in the left half
}
int main() {
    int n, key;
    printf("Enter the number of elements in the list: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d sorted elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the key value to search: ");
    scanf("%d", &key);
    int result_non_recursive = binarySearchNonRecursive(arr, n, key);
    if (result_non_recursive != -1) {
        printf("Key %d found at index %d (non-recursive).\n", key, result_non_recursive);
    } else {
        printf("Key %d not found (non-recursive).\n", key);
    }
```

```
    int result_recursive = binarySearchRecursive(arr, key, 0, n - 1);
    if (result_recursive != -1) {
        printf("Key %d found at index %d (recursive).\n", key, result_recursive);
    } else {
        printf("Key %d not found (recursive).\n", key);
    }
    return 0;
}
```

OUTPUT:

Experiment 7:

a) Write C program that implement stack (its operations) using arrays.

Source Code:

```
//implement a stack using array
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int stack[MAX];
int top = -1;
void push(int);
int pop();
void display();
int main()
{
   int ch, val;
   printf("\n1. Push in stack");
   printf("\n2. Pop from stack");
   printf("\n3. Display stack");
   printf("\n4. Exit");
   do
   {
      printf("\n\nEnter choice: ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         printf("\nEnter value to be pushed:");
         scanf("%d", &val);
         push(val);
         break;
      case 2:
         val = pop();
         if (val != -1)
            printf("\n The number deleted is : %d", val);
         break;
      case 3:
         display();
         break;
      case 4:
         printf("\nExit");
         break;
      default:
         printf("\nInvalid Choice");
      }
   } while (ch != 4);
   return 0;
```

```
}
void push(int val)
{
    if (top == MAX - 1)
        printf("\nStack is full");
    else
    {
        top++;
        stack[top] = val;
    }
}
int pop()
{
    int val;
    if (top == -1)
    {
        printf("\nStack is empty");
        return -1;
    }
    else
    {
        val = stack[top];
        top--;
        return val;
    }
}
void display()
{
    int i;
    if (top == -1)
        printf("\nStack is empty");
    else
    {
        printf("\nStack is...\n");
        for (i = top; i >= 0; i--)
            printf("%d\n", stack[i]);
    }
}
```

Output:

b) Write C program that implement stack (its operations) using Linked list.

Source Code:

```
//implemet stack using linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *top=NULL;
void push(int x)
{
   struct node *newnode;
   newnode=(struct node*)malloc(sizeof(struct node));
   newnode->data=x;
   newnode->next=top;
   top=newnode;
}
void pop()
{
   struct node *temp;
   if(top==NULL)
   {
      printf("stack is empty\n");
   }
   else
   {
      temp=top;//
      top=top->next;
      free(temp);
   }
}
void display()
{
   struct node *temp;
   if(top==NULL)
   {
      printf("stack is empty\n");
   }
   else
   {
      temp=top;
      while(temp!=NULL)
      {
         printf("%d\n",temp->data);
```

```
            temp=temp->next;
        }
    }
}
int main()
{
    int ch,x;
    while(1)
    {
        printf("1.push\n");
        printf("2.pop\n");
        printf("3.display\n");
        printf("4.exit\n");
        printf("enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
            printf("enter the element to be pushed\n");
            scanf("%d",&x);
            push(x);
            break;
            case 2:
            pop();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            default:
            printf("invalid choice\n");
        }
    }
    return 0;
}
```

Output:

Experiment 8:

a) Write a C program that uses Stack operations to convert infix expression into postfix expression.

Source Code:

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

#define MAX_SIZE 100

struct Stack {
    int top;
    char items[MAX_SIZE];
};

void push(struct Stack* stack, char item) {
    if (stack->top == MAX_SIZE - 1) {
        printf("Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->items[++(stack->top)] = item;
}

char pop(struct Stack* stack) {
    if (stack->top == -1) {
        printf("Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->items[(stack->top)--];
}

int isOperand(char ch) {
    return (ch >= '0' && ch <= '9') || (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}

int precedence(char ch) {
    if (ch == '+' || ch == '-') {
        return 1;
    } else if (ch == '*' || ch == '/') {
        return 2;
    } else if (ch == '^') {
        return 3;
    } else {
        return -1;
    }
}
```

```
void infixToPostfix(char infix[], char postfix[]) {
    struct Stack stack;
    stack.top = -1;
    int i, j;

    for (i = 0, j = 0; infix[i] != '\0'; i++) {
        if (isOperand(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(&stack, infix[i]);
        } else if (infix[i] == ')') {
            while (stack.top != -1 && stack.items[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            if (stack.top != -1 && stack.items[stack.top] != '(') {
                printf("Invalid Expression\n");
                exit(EXIT_FAILURE);
            } else {
                pop(&stack);
            }
        } else {
            while (stack.top != -1 && precedence(infix[i]) <= precedence(stack.items[stack.top])) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, infix[i]);
        }
    }

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

    postfix[j] = '\0';
}

int main() {
    char infix[MAX_SIZE], postfix[MAX_SIZE];

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

    infixToPostfix(infix, postfix);

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

    return 0;
```

```
}
```

Output:

b) Write C program that implement Queue (its operations) using arrays.

Source Code:

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

#define MAX_SIZE 10

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

int isFull() {
   return (rear == MAX_SIZE - 1);
}

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

void enqueue(int value) {
   if (isFull()) {
      printf("Queue Overflow\n");
      exit(EXIT_FAILURE);
   } else {
      if (isEmpty()) {
         front = rear = 0;
      } else {
         rear++;
      }
      queue[rear] = value;
   }
}

int dequeue() {
   int removedItem;

   if (isEmpty()) {
      printf("Queue Underflow\n");
      exit(EXIT_FAILURE);
   } else {
      removedItem = queue[front];
      if (front == rear) {
         front = rear = -1;
      } else {
         front++;
      }
   }
```

```
    return removedItem;
}

void display() {
    if (isEmpty()) {
        printf("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("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: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                if (!isEmpty()) {
                    printf("Dequeued element: %d\n", dequeue());
                }
                break;

            case 3:
                display();
                break;

            case 4:
                printf("Exiting...\n");
```

```
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Output:

c) Write C program that implement Queue (its operations) using linked lists.

Source Code:

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

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

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

int isEmpty() {
    return (front == NULL);
}

void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }

    newNode->data = value;
    newNode->next = NULL;

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

int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        exit(EXIT_FAILURE);
    }

    struct Node* temp = front;
    int removedItem = temp->data;

    front = temp->next;
    free(temp);
```

```
    if (front == NULL) {
        rear = NULL;
    }

    return removedItem;
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        struct Node* current = front;
        printf("Queue elements: ");
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}

int main() {
    int choice, value;

    do {
        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: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                if (!isEmpty()) {
                    printf("Dequeued element: %d\n", dequeue());
                }
                break;
```

```
            case 3:
                display();
                break;

            case 4:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Output:

Experiment 9:

Write a C program that uses functions to create a singly linked list and perform various operations on it.

Source Code:

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

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

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

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

void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        return;
    }

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

void deleteNode(struct Node** head, int value) {
    if (*head == NULL) {
        printf("List is empty, cannot delete\n");
```

```
        return;
    }

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

    if (temp != NULL && temp->data == value) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

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

    prev->next = temp->next;
    free(temp);
}

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

void freeList(struct Node** head) {
    struct Node* current = *head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *head = NULL;
```

```
}

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

    while (1) {
        printf("\n--- Menu ---\n");
        printf("1. Insert at the Beginning\n");
        printf("2. Insert at the End\n");
        printf("3. Delete Node\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(&head, value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(&head, value);
                break;
            case 3:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                deleteNode(&head, value);
                break;
            case 4:
                displayList(head);
                break;
            case 5:
                freeList(&head);
                printf("Exiting the program\n");
                exit(EXIT_SUCCESS);
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output:

Experiment 10:

Write a C program to store a polynomial expression in memory using linked list and perform polynomial addition.

Source Code:

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

struct Node {
    int coefficient;
    int exponent;
    struct Node* next;
};

struct Node* createNode(int coeff, int exp) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->coefficient = coeff;
    newNode->exponent = exp;
    newNode->next = NULL;
    return newNode;
}

void insertTerm(struct Node** poly, int coeff, int exp) {
    struct Node* newNode = createNode(coeff, exp);
    if (*poly == NULL) {
        *poly = newNode;
    } else {
        struct Node* current = *poly;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

void displayPolynomial(struct Node* poly) {
    struct Node* current = poly;
    while (current != NULL) {
        printf("%dx^%d", current->coefficient, current->exponent);
        current = current->next;
        if (current != NULL) {
            printf(" + ");
        }
```

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

struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;

    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {
            insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }

    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }

    return result;
}

void freeList(struct Node** poly) {
    struct Node* current = *poly;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *poly = NULL;
}
```

```
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    int coeff, exp;

    printf("Enter terms for Polynomial 1 (coeff exp, enter -1 -1 to stop):\n");
    while (1) {
        scanf("%d %d", &coeff, &exp);
        if (coeff == -1 && exp == -1) {
            break;
        }
        insertTerm(&poly1, coeff, exp);
    }

    printf("Enter terms for Polynomial 2 (coeff exp, enter -1 -1 to stop):\n");
    while (1) {
        scanf("%d %d", &coeff, &exp);
        if (coeff == -1 && exp == -1) {
            break;
        }
        insertTerm(&poly2, coeff, exp);
    }

    printf("\nPolynomial 1: ");
    displayPolynomial(poly1);

    printf("Polynomial 2: ");
    displayPolynomial(poly2);

    struct Node* result = addPolynomials(poly1, poly2);

    printf("\nSum of Polynomials: ");
    displayPolynomial(result);

    freeList(&poly1);
    freeList(&poly2);
    freeList(&result);

    return 0;
}
```

Output:

Experiment 11:

a) Write a recursive C program for traversing a binary tree in preorder, in order and post order.

Source Code:

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

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

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

struct Node* constructTree() {
    int value;
    printf("Enter data for a node (-1 for no node): ");
    scanf("%d", &value);

    if (value == -1) {
        return NULL;
    }

    struct Node* root = createNode(value);

    printf("Enter left child of %d\n", value);
    root->left = constructTree();

    printf("Enter right child of %d\n", value);
    root->right = constructTree();

    return root;
}

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

```
    preorderTraversal(root->right);
}

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

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

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

    printf("Construct the binary tree:\n");
    root = constructTree();

    printf("\nPreorder Traversal: ");
    preorderTraversal(root);

    printf("\nInorder Traversal: ");
    inorderTraversal(root);

    printf("\nPostorder Traversal: ");
    postorderTraversal(root);

    return 0;
}
```

Output:

b) Write a non-recursive C program for traversing a binary tree in preorder, in order and post order.

Source Code:

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

#define MAX_STACK_SIZE 100

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

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

struct Stack {
   int top;
   struct Node* items[MAX_STACK_SIZE];
};

struct Stack* createStack() {
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
   stack->top = -1;
   return stack;
}

void push(struct Stack* stack, struct Node* item) {
   if (stack->top == MAX_STACK_SIZE - 1) {
      printf("Stack Overflow\n");
      return;
   }
   stack->items[++(stack->top)] = item;
}

struct Node* pop(struct Stack* stack) {
   if (stack->top == -1) {
      printf("Stack Underflow\n");
      exit(EXIT_FAILURE);
   }
   return stack->items[(stack->top)--];
```

```
}

int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

struct Node* constructTree() {
    int value;
    printf("Enter data for a node (-1 for no node): ");
    scanf("%d", &value);

    if (value == -1) {
        return NULL;
    }

    struct Node* root = createNode(value);

    printf("Enter left child of %d\n", value);
    root->left = constructTree();

    printf("Enter right child of %d\n", value);
    root->right = constructTree();

    return root;
}

void preorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    struct Stack* stack = createStack();
    push(stack, root);

    while (!isEmpty(stack)) {
        struct Node* node = pop(stack);
        printf("%d ", node->data);

        if (node->right != NULL) {
            push(stack, node->right);
        }
        if (node->left != NULL) {
            push(stack, node->left);
        }
    }
}
```

```
void inorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    struct Stack* stack = createStack();
    struct Node* current = root;

    while (current != NULL || !isEmpty(stack)) {
        while (current != NULL) {
            push(stack, current);
            current = current->left;
        }
        current = pop(stack);
        printf("%d ", current->data);
        current = current->right;
    }
}

void postorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    struct Stack* stack1 = createStack();
    struct Stack* stack2 = createStack();
    push(stack1, root);

    while (!isEmpty(stack1)) {
        struct Node* node = pop(stack1);
        push(stack2, node);

        if (node->left != NULL) {
            push(stack1, node->left);
        }
        if (node->right != NULL) {
            push(stack1, node->right);
        }
    }

    while (!isEmpty(stack2)) {
        struct Node* node = pop(stack2);
        printf("%d ", node->data);
    }
}
```

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

    printf("Construct the binary tree:\n");
    root = constructTree();

    printf("\nPreorder Traversal: ");
    preorderTraversal(root);

    printf("\nInorder Traversal: ");
    inorderTraversal(root);

    printf("\nPostorder Traversal: ");
    postorderTraversal(root);

    return 0;
}
```

Output:

Experiment 12:

a) Write a C program to implement Prims' algorithm.

Source Code:

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

#define MAX_VERTICES 20

int minKey(int key[], int mstSet[], int vertices) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < vertices; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }

    return min_index;
}

void printMST(int parent[], int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int parent[MAX_VERTICES];
    int key[MAX_VERTICES];
    int mstSet[MAX_VERTICES];

    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < vertices - 1; count++) {
        int u = minKey(key, mstSet, vertices);
        mstSet[u] = 1;

        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
```

```
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, vertices);
}

int main() {
    int vertices;

    printf("Enter the number of vertices (maximum %d): ", MAX_VERTICES);
    scanf("%d", &vertices);

    int graph[MAX_VERTICES][MAX_VERTICES];

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    primMST(graph, vertices);

    return 0;
}
```

Output:

b) Write a C program to implement Kruskal's algorithm.

Source Code:

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

#define MAX_VERTICES 20
#define MAX_EDGES 50

struct Edge {
    int src, dest, weight;
};

struct Subset {
    int parent, rank;
};

int find(struct Subset subsets[], int i);
void Union(struct Subset subsets[], int x, int y);
int compare(const void* a, const void* b);
void kruskalMST(struct Edge edges[], int V, int E);

int main() {
    int V, E;

    printf("Enter the number of vertices (maximum %d): ", MAX_VERTICES);
    scanf("%d", &V);

    printf("Enter the number of edges (maximum %d): ", MAX_EDGES);
    scanf("%d", &E);

    struct Edge edges[MAX_EDGES];

    printf("Enter the edges (source destination weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }

    kruskalMST(edges, V, E);

    return 0;
}

int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
```

```
    return subsets[i].parent;
}

void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int compare(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}

void kruskalMST(struct Edge edges[], int V, int E) {
    qsort(edges, E, sizeof(edges[0]), compare);

    struct Subset subsets[V];
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    struct Edge result[V];
    int e = 0, i = 0;

    while (e < V - 1 && i < E) {
        struct Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Edge \tWeight\n");
```

```
    for (i = 0; i < e; i++)
        printf("%d - %d \t%d \n", result[i].src, result[i].dest, result[i].weight);
}
```

Output:

Experiment 13:

Implementation of Hash table using double hashing as collision resolution function.

Source Code:

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

#define SIZE 10

struct HashNode {
    int key;
    int value;
};

struct HashTable {
    struct HashNode* array;
    int size;
};

struct HashTable* createHashTable(int size) {
    struct HashTable* table = (struct HashTable*)malloc(sizeof(struct HashTable));
    if (table == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }

    table->size = size;
    table->array = (struct HashNode*)malloc(size * sizeof(struct HashNode));

    for (int i = 0; i < size; i++) {
        table->array[i].key = -1;
        table->array[i].value = 0;
    }

    return table;
}

int hashFunction1(int key, int size) {
    return key % size;
}

int hashFunction2(int key) {
    return (key % (SIZE - 1)) + 1;
}

int doubleHashing(int key, int i, int size) {
    return (hashFunction1(key, size) + i * hashFunction2(key)) % size;
```

```
}

void insert(struct HashTable* table, int key, int value) {
    int index = hashFunction1(key, table->size);
    int i = 0;

    while (table->array[index].key != -1) {
        index = doubleHashing(key, i, table->size);
        i++;
    }

    table->array[index].key = key;
    table->array[index].value = value;
}

int search(struct HashTable* table, int key) {
    int index = hashFunction1(key, table->size);
    int i = 0;

    while (table->array[index].key != key && table->array[index].key != -1) {
        index = doubleHashing(key, i, table->size);
        i++;
    }

    if (table->array[index].key == key) {
        return table->array[index].value;
    } else {
        return -1;
    }
}

void displayHashTable(struct HashTable* table) {
    printf("Hash Table:\n");
    for (int i = 0; i < table->size; i++) {
        if (table->array[i].key != -1) {
            printf("Index: %d, Key: %d, Value: %d\n", i, table->array[i].key, table->array[i].value);
        }
    }
}

int main() {
    int size;

    printf("Enter the size of the hash table: ");
    scanf("%d", &size);
```

```
    struct HashTable* hashTable = createHashTable(size);

    int key, value;

    for (int i = 0; i < size; i++) {
        printf("Enter key and value for position %d (key -1 to stop): ", i);
        scanf("%d", &key);

        if (key == -1) {
            break;
        }

        scanf("%d", &value);
        insert(hashTable, key, value);
    }

    displayHashTable(hashTable);

    int keyToSearch;
    printf("Enter key to search: ");
    scanf("%d", &keyToSearch);

    int result = search(hashTable, keyToSearch);

    if (result != -1) {
        printf("Value for key %d: %d\n", keyToSearch, result);
    } else {
        printf("Key %d not found in the hash table\n", keyToSearch);
    }

    free(hashTable->array);
    free(hashTable);

    return 0;
}
```

Output:

Experiment 14:

Implementation of Binary Search trees- Insertion and deletion.

Source Code:

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

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

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

struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

struct Node* deleteNode(struct Node* root, int value) {
    if (root == NULL) {
        return root;
    }
```

```
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

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

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

    int choice, value;

    do {
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display (Inorder Traversal)\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);
                root = insert(root, value);
                break;

            case 2:
                printf("Enter the value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;

            case 3:
                printf("Inorder Traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;

            case 4:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Output:

Experiment 15:

a) Write C program that implement Bubble sort, to sort a given list of integers in ascending order.

Source Code:

```
#include<stdio.h>
void BubbleSort(int A[], int n){
    int cnt = 0;
    for(int i = 0 ; i < n - 1 ; i++){
        int flage = 0;
        for(int j = 0 ; j < n- i -1 ; j++){
            cnt++;
            if(A[j] > A[j+1]){
                int temp = A[j+1];
                A[j+1] = A[j];
                A[j] = temp;
                flage = 1;//
            }
        }
        if(flage == 0){
            break;
        }
    }
    printf("%d\n",cnt);
}
int main(){
    int n;
    scanf("%d", &n);
    int A[n];
    for(int i=0; i<n; i++){
        scanf("%d", &A[i]);
    }
    BubbleSort(A,n);
    //time complexity of bubble sort is O(n^2)

    //array will be sorted
    for(int i = 0 ; i < n ; i++){
        printf("%d ", A[i]);
    }
}
```

Output:

b) Write C program that implement Quick sort, to sort a given list of integers in ascending order.

Source Code:

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

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

int main() {
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);
```

```
    int arr[size];

    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

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

    quickSort(arr, 0, size - 1);

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

    return 0;
}
```

Output:

c) Write C program that implement merge sort, to sort a given list of integers in ascending order.

Source Code:

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int leftArray[n1], rightArray[n2];

    for (int i = 0; i < n1; i++) {
        leftArray[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArray[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            i++;
        } else {
            arr[k] = rightArray[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArray[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = rightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
```

```
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

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

int main() {
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];

    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

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

    mergeSort(arr, 0, size - 1);

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

    return 0;
}
```

Output: