1.Write a C program to perform Matrix Multiplication

```c
#include <stdio.h>

#define N 3

void multiplyMatrix(int firstMatrix[][N], int secondMatrix[][N], int result[][N]) {

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

        for (int j = 0; j < N; j++) {

            result[i][j] = 0;

            for (int k = 0; k < N; k++) {

                result[i][j] += firstMatrix[i][k] * secondMatrix[k][j];

            }

        }

    }

}

int main() {

    int firstMatrix[N][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    int secondMatrix[N][N] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

    int result[N][N];

multiplyMatrix(firstMatrix, secondMatrix, result);

  printf("Result of Matrix Multiplication:\n");

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

        for (int j = 0; j < N; j++) {

            printf("%d ", result[i][j]);

        }

        printf("\n");

    }

    return 0; }
```

```
Result of Matrix Multiplication:
30 24 18
84 69 54
138 114 90
```

2.Write a C program to search a number using Linear Search method

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {

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

      if (arr[i] == key) {

         return i;

      }

   }

   return -1;

}

int main() {

   int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};

   int n = sizeof(arr) / sizeof(arr[0]);

   int key = 23;

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

   if (result != -1) {

      printf("Element found at index: %d", result);

   } else {

      printf("Element not found");

   }

   return 0;

}
```

```
Element found at index: 5
```

3.Write a C program to search a number using Binary Search method

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) {

    while (left <= right) {

        int mid = left + (right - left) / 2;

   if (arr[mid] == target)

            return mid;

 if (arr[mid] < target)

            left = mid + 1;

        else

            right = mid - 1;

    }

return -1;

}

int main() {

    int arr[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    int target = 12;

    int result = binarySearch(arr, 0, n - 1, target);

    if (result == -1)

        printf("Element not found\n");

    else

        printf("Element found at index %d\n", result);

  return 0;

}
```

```
Element found at index: 5
```

4.Write a C program to implement the Tree Traversals (In order, Preorder, Post order)

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node* left;

    struct Node* right;

};

struct Node* createNode(int data) {

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

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void inOrder(struct Node* root) {

    if (root == NULL)

        return;

    inOrder(root->left);

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

    inOrder(root->right);

}

void preOrder(struct Node* root) {

    if (root == NULL)

        return;

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

    preOrder(root->left);

    preOrder(root->right);

}
```

```
void postOrder(struct Node* root) {
    if (root == NULL)
        return;
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    printf("Inorder traversal: ");
    inOrder(root);
    printf("\n");
    printf("Preorder traversal: ");
    preOrder(root);
    printf("\n");

   printf("Postorder traversal: ");
    postOrder(root);
    printf("\n");
    return 0;
}
```

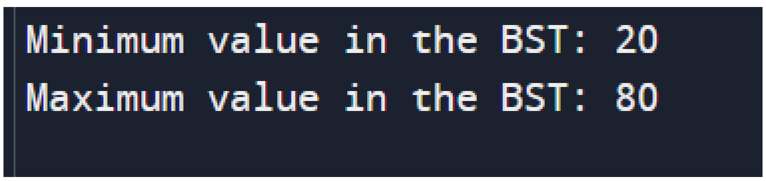
```
Inorder traversal: 4 2 5 1 3
Preorder traversal: 1 2 4 5 3
Postorder traversal: 4 5 2 3 1
```

5. Write a C program to search for a number, Min, Max from a BST

```
#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;
```

```
}

int findMin(struct Node* root) {
    if (root == NULL) {
        printf("Error: Tree is empty\n");
        return -1;
    }

    while (root->left != NULL) {
        root = root->left;
    }

    return root->data;
}

int findMax(struct Node* root) {
    if (root == NULL) {
        printf("Error: Tree is empty\n");
        return -1;
    }

    while (root->right != NULL) {
        root = root->right;
    }

    return root->data;
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
```

```
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Minimum value in the BST: %d\n", findMin(root));
    printf("Maximum value in the BST: %d\n", findMax(root));

    return 0;
}
```



```
Minimum value in the BST: 20
Maximum value in the BST: 80
```

6.Write a program in C to read n number of values in an array and display them in reverse order.

```
#include <stdio.h>
int main() {
   int n, i;
    printf("Input the number of elements to store in the array: ");
   scanf("%d", &n);
    int arr[n];
   printf("Input %d number of elements in the array:\n", n);
    for (i = 0; i < n; i++) {
      printf("element - %d : ", i);
      scanf("%d", &arr[i]);
   }
 printf("The values stored into the array are:\n");
   for (i = 0; i < n; i++) {
```

```
        printf("%d ", arr[i]);
    }
   printf("\nThe values stored into the array in reverse are: ");
     for (i = n - 1; i >= 0; i--) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

```
Input the number of elements to store in the array: 3
Input 3 number of elements in the array:
element - 0 : 2
element - 1 : 5
element - 2 : 7
The values stored into the array are:
2 5 7
The values stored into the array in reverse are: 7 5 2

=== Code Execution Successful ===
```

7. Implement a C Program for AVL tree and perform Insertion and Deletion of nodes

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

// AVL tree node
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
```

```
    int height;
};

// Function to get the height of the tree
int height(struct Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Helper function to create a new node
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;  // new node is initially added at leaf
    return(node);
}

// Right rotate subtree rooted with y
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;
```

```
    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}
```

```
// Get balance factor of node N
int getBalance(struct Node* N) {
   if (N == NULL)
      return 0;
   return height(N->left) - height(N->right);
}

// Insert a node
struct Node* insert(struct Node* node, int key) {
   // 1. Perform the normal BST rotation
   if (node == NULL)
      return(newNode(key));

   if (key < node->key)
      node->left = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
   else  // Equal keys are not allowed in BST
      return node;

   // 2. Update height of this ancestor node
   node->height = 1 + max(height(node->left), height(node->right));

   // 3. Get the balance factor of this ancestor node to check whether
   //    this node became unbalanced
   int balance = getBalance(node);

   // If this node becomes unbalanced, then there are 4 cases
```

```
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // return the (unchanged) node pointer
    return node;
}

// Helper function to find the node with the minimum key value
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
```

```
    // loop down to find the leftmost leaf
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Delete a node
struct Node* deleteNode(struct Node* root, int key) {
    // 1. Perform standard BST delete
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node* temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else  // One child case
                *root = *temp;  // Copy the contents of the non-empty child
```

```
            free(temp);
        }
        else {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct Node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // 2. Update height of the current node
    root->height = 1 + max(height(root->left), height(root->right));

    // 3. Get the balance factor of this node (to check whether this node
    //    became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases
```

```
    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of the tree
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
```

```c
    }
}

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

    /* Constructing tree */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
            30
           /  \
         20   40
        /  \    \
      10   25   50
    */

    printf("Preorder traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
```

```
        30
       / \
     20   40
       \   \
       25   50
  */

  printf("\nPreorder traversal after deletion of 10 \n");
  preOrder(root);

  return 0;
}
```

```bash
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50

Preorder traversal after deletion of 10
30 20 25 40 50
```

8. Implement a C Program to Check for a valid String using stack

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

```
#define MAX 100

typedef struct Stack {
    char arr[MAX];
    int top;
} Stack;

void initStack(Stack* s) {
    s->top = -1;
}

int isFull(Stack* s) {
    return s->top == MAX - 1;
}

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

void push(Stack* s, char c) {
```

```
    if (!isFull(s)) {
        s->arr[++s->top] = c;
    }
}

char pop(Stack* s) {
    if (!isEmpty(s)) {
        return s->arr[s->top--];
    }
    return '\0';
}

int isValidString(const char* str) {
    Stack s;
    initStack(&s);

    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == '(' || str[i] == '{' || str[i] == '[') {
            push(&s, str[i]);
        } else if (str[i] == ')' || str[i] == '}' || str[i] == ']') {
```

```
            char top = pop(&s);
            if ((str[i] == ')' && top != '(') ||
                (str[i] == '}' && top != '{') ||
                (str[i] == ']' && top != '[')) {
                return 0;
            }
        }
    }
    return isEmpty(&s);
}

int main() {
    const char* testString = "{[()]}";
    if (isValidString(testString)) {
        printf("The string is valid.\n");
    } else {
        printf("The string is invalid.\n");
    }
    return 0;
}
```

```
The string is valid.


=== Code Execution Successful ===
```

.Write a program in C to count the total number of duplicate elements in an array.

Test Data :

Input the number of elements to be stored in the array :3

Input 3 elements in the array :

element - 0 : 5

element - 1 : 1

element - 2 : 1

Expected Output :

Total number of duplicate elements found in the array is : 1

```
#include <stdio.h>

int main() {

   int n, i, j, count = 0;

   printf("Input the number of elements to be stored in the array: ");

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

```
    int arr[n];
    printf("Input %d elements in the array:\n", n);
    for(i = 0; i < n; i++) {
        printf("element - %d : ", i);
        scanf("%d", &arr[i]);
    }
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) {
            if(arr[i] == arr[j]) {
                count++;
                break;
            }
        }
    }
  printf("Total number of duplicate elements found in the array
is: %d\n", count);
    return 0;
}
```

```
Input the number of elements to be stored in the array: 3
Input 3 elements in the array:
element - 0 : 5
element - 1 : 1
element - 2 : 1
Total number of duplicate elements found in the array is: 1


=== Code Execution Successful ===
```

10.Implement a C Program for Merging of list

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* mergeLists(struct Node* list1, struct Node* list2) {
    if (!list1) return list2;
    if (!list2) return list1;
    if (list1->data < list2->data) {
        list1->next = mergeLists(list1->next, list2);
        return list1;
```

```
    } else {
        list2->next = mergeLists(list1, list2->next);
        return list2;
    }
}
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct
Node));
    node->data = data;
    node->next = NULL;
    return node;
}
int main() {
```

```
    struct Node* list1 = newNode(1);
    list1->next = newNode(3);
    list1->next->next = newNode(5);
    struct Node* list2 = newNode(2);
    list2->next = newNode(4);
    list2->next->next = newNode(6);
    struct Node* mergedList = mergeLists(list1, list2);
    printList(mergedList);
    return 0;
}
```

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL


=== Code Execution Successful ===
```

Write a program in C to count the frequency of each element of an array.

Test Data :

Input the number of elements to be stored in the array :3

Input 3 elements in the array :

element - 0 : 25

element - 1 : 12

element - 2 : 43

Expected Output :

The frequency of all element

```
#include <stdio.h>

int main() {
   int n, i, j, count;
   printf("Input the number of elements to be stored in the
array: ");
   scanf("%d", &n);

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

   printf("The frequency of all elements:\n");
   for(i = 0; i < n; i++) {
```

```
        count = 1;
        for(j = i + 1; j < n; j++) {
            if(arr[i] == arr[j]) {
                count++;
                for(int k = j; k < n - 1; k++) {
                    arr[k] = arr[k + 1];
                }
                n--;
                j--;
            }
        }
        printf("%d occurs %d times\n", arr[i], count);
    }

    return 0;
}
```

```
Input the number of elements to be stored in the array: 3
Input 3 elements in the array:
element - 0 : 25
element - 1 : 12
element - 2 : 43
The frequency of all elements:
25 occurs 1 times
12 occurs 1 times
43 occurs 1 times


=== Code Execution Successful ===
```