🌳 Clean BST Program with Traversals

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

// Structure for a node
struct Node {
    int data;
    struct Node *left, *right;
};

// Function to create a new node
struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a node
struct Node* insert(struct Node* root, int val) {
    if (root == NULL)
        return createNode(val);

    if (val < root->data)
        root->left = insert(root->left, val);
    else if (val > root->data)
        root->right = insert(root->right, val);

    return root;
}

// Inorder Traversal (Left → Root → Right)
void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder Traversal (Root → Left → Right)
void preorder(struct Node* root) {
    if (root) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder Traversal (Left → Right → Root)
void postorder(struct Node* root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Find minimum value node in a subtree
struct Node* findMin(struct Node* root) {
    while (root->left)
        root = root->left;
    return root;
}

// Delete a node from the BST
struct Node* deleteNode(struct Node* root, int val) {
    if (root == NULL)
        return NULL;

    if (val < root->data)
        root->left = deleteNode(root->left, val);
    else if (val > root->data)
        root->right = deleteNode(root->right, val);
    else {
        // Node found
        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;
        }

        // Node with two children
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

```c
// Main function
int main() {
    struct Node* root = NULL;
    int ch, val;

    do {
        printf("\n--- Binary Search Tree Menu ---\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Inorder Traversal\n");
        printf("4. Preorder Traversal\n");
        printf("5. Postorder Traversal\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &val);
                root = insert(root, val);
                break;

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

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

            case 4:
                printf("Preorder: ");
                preorder(root);
                printf("\n");
                break;

            case 5:
                printf("Postorder: ");
                postorder(root);
                printf("\n");
                break;

            case 6:
                printf("Exiting program...\n");
                break;

            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (ch != 6);

    return 0;
}
```

🌐 Graph Traversal using DFS & BFS

```
#include <stdio.h>
#define MAX 10

int adj[MAX][MAX], visited[MAX], n;

// Depth First Search (DFS)
void DFS(int v) {
    printf("%d ", v);
    visited[v] = 1;

    for (int i = 0; i < n; i++) {
        if (adj[v][i] && !visited[i]) {
            DFS(i);
        }
    }
}

// Breadth First Search (BFS)
void BFS(int start) {
    int q[MAX], front = 0, rear = 0;

    // Reset visited array
    for (int i = 0; i < n; i++)
        visited[i] = 0;

    printf("%d ", start);
    visited[start] = 1;
    q[rear++] = start;

    while (front != rear) {
        int v = q[front++];
```

```c
        for (int i = 0; i < n; i++) {
            if (adj[v][i] && !visited[i]) {
                printf("%d ", i);
                visited[i] = 1;
                q[rear++] = i;
            }
        }
    }
}

// Main function
int main() {
    int ch, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

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

    do {
        printf("\n--- Graph Traversal Menu ---\n");
        printf("1. Depth First Search (DFS)\n");
        printf("2. Breadth First Search (BFS)\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                for (int i = 0; i < n; i++)
                    visited[i] = 0;
                printf("Enter starting vertex: ");
                scanf("%d", &start);
                printf("DFS Traversal: ");
                DFS(start);
                printf("\n");
                break;

            case 2:
                printf("Enter starting vertex: ");
                scanf("%d", &start);
                printf("BFS Traversal: ");
                BFS(start);
                printf("\n");
                break;

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

            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (ch != 3);

    return 0;
}
```

🧠 Program: Array Operations (Static & Dynamic Memory Allocation)

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

// Structure for Node
struct Node {
    int data;
    struct Node *next;
};

struct Node *head = NULL;

// 🌱 Create Node
struct Node* createNode(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = NULL;
    return n;
}

// 🟢 Insert at Front
void insertFront(int val) {
    struct Node *n = createNode(val);
    n->next = head;
    head = n;
}

// 🔵 Insert at Rear
void insertRear(int val) {
    struct Node *n = createNode(val);
    if (!head) {
```

```c
        head = n;
        return;
    }
    struct Node *t = head;
    while (t->next)
        t = t->next;
    t->next = n;
}

// 🟡 Insert by Position
void insertPos(int val, int pos) {
    struct Node *n = createNode(val);
    if (pos == 1) {
        n->next = head;
        head = n;
        return;
    }
    struct Node *t = head;
    int i = 1;
    while (t && i < pos - 1) {
        t = t->next;
        i++;
    }
    if (!t) {
        printf("Invalid Position!\n");
        free(n);
        return;
    }
    n->next = t->next;
    t->next = n;
}

// 🔴 Delete Front
void deleteFront() {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }
    struct Node *t = head;
    head = head->next;
    free(t);
}

// 🟠 Delete Rear
void deleteRear() {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }
    if (!head->next) {
        free(head);
        head = NULL;
        return;
    }
    struct Node *t = head;
    while (t->next->next)
        t = t->next;
    free(t->next);
    t->next = NULL;
}

// 🟣 Delete by Position
void deletePos(int pos) {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }
    if (pos == 1) {
        deleteFront();
        return;
    }
    struct Node *t = head;
    int i = 1;
    while (t->next && i < pos - 1) {
        t = t->next;
        i++;
    }
    if (!t->next) {
        printf("Invalid Position!\n");
        return;
    }
    struct Node *temp = t->next;
    t->next = temp->next;
    free(temp);
}

// ⚫ Delete by Key
void deleteKey(int key) {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }
    if (head->data == key) {
        deleteFront();
        return;
    }
    struct Node *t = head;
    while (t->next && t->next->data != key)
        t = t->next;
    if (!t->next) {
        printf("Key Not Found!\n");
        return;
    }
    struct Node *temp = t->next;
    t->next = temp->next;
    free(temp);
}

// 🧩 Search by Key
void searchKey(int key) {
    struct Node *t = head;
    int pos = 1;
    while (t) {
        if (t->data == key) {
            printf("Key %d found at position %d\n", key, pos);
            return;
        }
        t = t->next;
        pos++;
    }
    printf("Key Not Found!\n");
}

// 🪜 Insert in Ordered (Ascending)
void insertOrdered(int val) {
    struct Node *n = createNode(val);
    if (!head || val < head->data) {
        n->next = head;
        head = n;
        return;
    }
    struct Node *t = head;
    while (t->next && t->next->data < val)
        t = t->next;
    n->next = t->next;
    t->next = n;
}

// 🔁 Reverse List
void reverseList() {
    struct Node *prev = NULL, *cur = head, *next = NULL;
    while (cur) {
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    }
    head = prev;
}

// 🧬 Copy List
struct Node* copyList() {
    if (!head)
        return NULL;

    struct Node *src = head, *copy = NULL, *tail = NULL;
    while (src) {
        struct Node *n = createNode(src->data);
        if (!copy)
            copy = tail = n;
        else {
            tail->next = n;
            tail = n;
        }
        src = src->next;
    }
    printf("List Copied Successfully!\n");
    return copy;
}

// 🖥️ Display List
void display(struct Node *h) {
    if (!h) {
        printf("List is Empty!\n");
        return;
    }
    while (h) {
        printf("%d -> ", h->data);
        h = h->next;
    }
    printf("NULL\n");
}

// 🧾 Main Function
int main() {
    int ch, val, pos, key;
    struct Node *copy = NULL;

    do {
        printf("\n--- Linked List Operations ---\n");
        printf("1. Insert Front\n");
        printf("2. Insert Rear\n");
        printf("3. Insert by Position\n");
        printf("4. Delete Front\n");
        printf("5. Delete Rear\n");
        printf("6. Delete by Position\n");
        printf("7. Delete by Key\n");
        printf("8. Search by Key\n");
        printf("9. Insert Ordered (Ascending)\n");
        printf("10. Reverse List\n");
        printf("11. Copy List\n");
        printf("12. Display List\n");
        printf("13. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                insertFront(val);
                break;

            case 2:
                printf("Enter value: ");
                scanf("%d", &val);
                insertRear(val);
                break;

            case 3:
                printf("Enter value and position: ");
                scanf("%d%d", &val, &pos);
                insertPos(val, pos);
                break;

            case 4:
                deleteFront();
                break;

            case 5:
                deleteRear();
                break;

            case 6:
                printf("Enter position: ");
                scanf("%d", &pos);
                deletePos(pos);
                break;

            case 7:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                deleteKey(key);
                break;

            case 8:
                printf("Enter key to search: ");
                scanf("%d", &key);
                searchKey(key);
                break;

            case 9:
                printf("Enter value to insert in order: ");
                scanf("%d", &val);
                insertOrdered(val);
                break;

            case 10:
                reverseList();
                printf("List Reversed!\n");
                break;

            case 11:
                copy = copyList();
                display(copy);
                break;

            case 12:
                display(head);
                break;

            case 13:
                printf("Exiting program...\n");
                break;

            default:
                printf("Invalid Choice! Try Again.\n");
        }
    } while (ch != 13);

    return 0;
}
```

💫 Singly Linked List – Menu Driven Program 💫

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

```c
// 🌿 Node structure
struct Node {
    int data;
    struct Node *next;
};

struct Node *head = NULL;

// 🌱 Insert at Front
void insertFront(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = head;
    head = n;
}

// 🌻 Insert at Rear
void insertRear(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = NULL;

    if (!head) {
        head = n;
    } else {
        struct Node *t = head;
        while (t->next)
            t = t->next;
        t->next = n;
    }
}

// 🌷 Insert at Specific Position
void insertPos(int val, int pos) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;

    if (pos == 1) {
        n->next = head;
        head = n;
        return;
    }

    struct Node *t = head;
    int i;
    for (i = 1; i < pos - 1 && t; i++)
        t = t->next;
    if (!t) {
        printf("Invalid position!\n");
        free(n);
        return;
    }

    n->next = t->next;
    t->next = n;
}

// 🍂 Delete Front Node
void deleteFront() {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }

    struct Node *t = head;
    head = head->next;
    free(t);
}

// 🍁 Delete Rear Node
void deleteRear() {
    if (!head) {
        printf("List is Empty!\n");
        return;
    }

    if (!head->next) {
        free(head);
        head = NULL;
        return;
    }

    struct Node *t = head, *prev = NULL;
    while (t->next) {
        prev = t;
        t = t->next;
    }

    prev->next = NULL;
    free(t);
}

// 🍀 Delete by Key
void deleteKey(int key) {
    struct Node *t = head, *prev = NULL;

    while (t && t->data != key) {
        prev = t;
        t = t->next;
    }

    if (!t) {
        printf("Key not found!\n");
        return;
    }

    if (!prev)
        head = head->next;
    else
        prev->next = t->next;

    free(t);
}

// 🌼 Search by Key
void searchKey(int key) {
    struct Node *t = head;
    int pos = 1;

    while (t) {
        if (t->data == key) {
            printf("Key found at position %d\n", pos);
            return;
        }
        t = t->next;
        pos++;
    }

    printf("Key not found!\n");
}

// 🌸 Insert in Sorted Order
void insertOrdered(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = NULL;

    if (!head || val < head->data) {
        n->next = head;
        head = n;
        return;
    }
    struct Node *t = head;
    while (t->next && t->next->data < val)
        t = t->next;

    n->next = t->next;
    t->next = n;
}

// 🌺 Reverse the List
void reverseList() {
    struct Node *prev = NULL, *cur = head, *next = NULL;

    while (cur) {
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    }

    head = prev;
    printf("List reversed successfully!\n");
}

// 🍇 Copy the List
struct Node* copyList() {
    if (!head) return NULL;

    struct Node *copy = NULL, *tail = NULL, *t = head;

    while (t) {
        struct Node *n = (struct Node*)malloc(sizeof(struct Node));
        n->data = t->data;
        n->next = NULL;

        if (!copy)
            copy = tail = n;
        else {
            tail->next = n;
            tail = n;
        }

        t = t->next;
    }

    printf("List copied successfully!\n");
    return copy;
}
```

```c
// 🌻 Display List
void display(struct Node *h) {
    if (!h) {
        printf("List is Empty!\n");
        return;
    }

    while (h) {
        printf("%d -> ", h->data);
        h = h->next;
    }

    printf("NULL\n");
}

// 🧠 Main Function -- Menu Driven
int main() {
    int ch, val, pos, key;
    struct Node *copy = NULL;

    do {
        printf("\n-------------------- MENU --------------------\n");
        printf("1. Insert Front\n");
        printf("2. Insert Rear\n");
        printf("3. Insert at Position\n");
        printf("4. Delete Front\n");
        printf("5. Delete Rear\n");
        printf("6. Delete by Key\n");
        printf("7. Search by Key\n");
        printf("8. Insert in Order\n");
        printf("9. Reverse List\n");
        printf("10. Copy List\n");
        printf("11. Display\n");
        printf("12. Exit\n");
        printf("----------------------------------------------\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                insertFront(val);
                break;

            case 2:
                printf("Enter value: ");
                scanf("%d", &val);
                insertRear(val);
                break;

            case 3:
                printf("Enter value and position: ");
                scanf("%d%d", &val, &pos);
                insertPos(val, pos);
                break;

            case 4:
                deleteFront();
                break;

            case 5:
                deleteRear();
                break;

            case 6:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                deleteKey(key);
                break;

            case 7:
                printf("Enter key to search: ");
                scanf("%d", &key);
                searchKey(key);
                break;

            case 8:
                printf("Enter value: ");
                scanf("%d", &val);
                insertOrdered(val);
                break;

            case 9:
                reverseList();
                break;

            case 10:
                copy = copyList();
                printf("Copied List: ");
                display(copy);
                break;

            case 11:
                display(head);
```

```c
                break;

            case 12:
                printf("Exiting program... 👋\n");
                break;

            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (ch != 12);

    return 0;
}
```

circular Singly Linked List (with Header Node) 👇

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

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

struct Node *head = NULL;

// 🌟 Create header node
void createHeader() {
    head = (struct Node*)malloc(sizeof(struct Node));
    head->data = -1;
    head->next = head;
}

// 🟢 Insert at front
void insertFront(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = head->next;
    head->next = n;
}

// 🔵 Insert at rear
void insertRear(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    struct Node *t = head;
    while (t->next != head)
        t = t->next;
    n->next = head;
    t->next = n;
}

// 🟣 Insert by position
void insertPos(int val, int pos) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    struct Node *t = head;
    int i = 0;
    while (t->next != head && i < pos - 1) {
        t = t->next;
        i++;
    }
    n->next = t->next;
    t->next = n;
}

// 🔴 Delete front
void deleteFront() {
    if (head->next == head) {
        printf("List is Empty!\n");
        return;
    }
    struct Node *t = head->next;
    head->next = t->next;
    free(t);
}

// 🟠 Delete rear
void deleteRear() {
    if (head->next == head) {
        printf("List is Empty!\n");
        return;
    }
    struct Node *t = head, *prev = NULL;
    while (t->next != head) {
        prev = t;
        t = t->next;
    }
    prev->next = head;
    free(t);
}

// 🟡 Delete by position
void deletePos(int pos) {
    if (head->next == head) {
        printf("List is Empty!\n");
        return;
    }
    struct Node *t = head;
    int i = 0;
    while (t->next != head && i < pos - 1) {
        t = t->next;
        i++;
    }
    if (t->next == head) {
        printf("Invalid Position!\n");
        return;
    }
    struct Node *temp = t->next;
    t->next = temp->next;
    free(temp);
}

// 🧩 Delete by key
void deleteKey(int key) {
    struct Node *t = head;
    while (t->next != head && t->next->data != key)
        t = t->next;
    if (t->next == head) {
        printf("Key not found!\n");
        return;
    }
    struct Node *temp = t->next;
    t->next = temp->next;
    free(temp);
}

// 🔍 Search by key
void searchKey(int key) {
    struct Node *t = head->next;
    int pos = 1;
    while (t != head) {
        if (t->data == key) {
            printf("Key %d found at position %d\n", key, pos);
            return;
        }
        t = t->next;
        pos++;
    }
    printf("Key not found!\n");
}
```

```c
// 🧮 Ordered insert
void insertOrdered(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    struct Node *t = head;
    while (t->next != head && t->next->data < val)
        t = t->next;
    n->next = t->next;
    t->next = n;
}

// 🔁 Reverse list
void reverseList() {
    struct Node *prev = head, *cur = head->next, *next;
    if (cur == head)
        return;
    do {
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    } while (cur != head);
    head->next = prev;
    printf("List reversed!\n");
}

// 📋 Copy list
struct Node* copyList() {
    struct Node *copyHead = (struct Node*)malloc(sizeof(struct Node));
    copyHead->data = -1;
    copyHead->next = copyHead;
    struct Node *src = head->next, *last = copyHead;
    while (src != head) {
        struct Node *n = (struct Node*)malloc(sizeof(struct Node));
        n->data = src->data;
        n->next = copyHead;
        last->next = n;
        last = n;
        src = src->next;
    }
    printf("List copied!\n");
    return copyHead;
}

// 👁️ Display
void display(struct Node *h) {
    if (h->next == h) {
        printf("List is Empty!\n");
        return;
    }
    struct Node *t = h->next;
    while (t != h) {
        printf("%d -> ", t->data);
        t = t->next;
    }
    printf("(Back to Head)\n");
}

// 🧠 Main function
int main() {
    int ch, val, pos, key;
    struct Node *copy = NULL;
    createHeader();

    do {
        printf("\n========= Circular Singly Linked List Menu =========\n");
        printf("1.Insert Front  2.Insert Rear  3.Insert Position\n");
        printf("4.Delete Front  5.Delete Rear  6.Delete Position\n");
        printf("7.Delete Key    8.Search Key   9.Ordered Insert\n");
        printf("10.Reverse List 11.Copy List  12.Display  13.Exit\n");
        printf("=====================================================\n");
        printf("Enter choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                insertFront(val);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &val);
                insertRear(val);
                break;
            case 3:
                printf("Enter value and position: ");
                scanf("%d%d", &val, &pos);
                insertPos(val, pos);
                break;
            case 4:
                deleteFront();
                break;
            case 5:
                deleteRear();
                break;
            case 6:
                printf("Enter position: ");
                scanf("%d", &pos);
                deletePos(pos);
                break;
            case 7:
                printf("Enter key: ");
                scanf("%d", &key);
                deleteKey(key);
                break;
            case 8:
                printf("Enter key: ");
                scanf("%d", &key);
                searchKey(key);
                break;
            case 9:
                printf("Enter value: ");
                scanf("%d", &val);
                insertOrdered(val);
                break;
            case 10:
                reverseList();
                break;
            case 11:
                copy = copyList();
                display(copy);
                break;
            case 12:
                display(head);
                break;
            case 13:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while (ch != 13);

    return 0;
}
```

Circular Doubly Linked List (with Header Node) 👇

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

```c
// 🧩 Node structure
struct Node {
    int data;
    struct Node *next;
    struct Node *prev;
};

// Header node pointer
struct Node *head = NULL;

// 🌟 Create header node
void createHeader() {
    head = (struct Node*)malloc(sizeof(struct Node));
    head->data = -1; // header node value
    head->next = head;
    head->prev = head;
}

// 🟢 Insert at Front
void insertFront(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = head->next;
    n->prev = head;
    head->next->prev = n;
    head->next = n;
}

// 🔵 Insert at Rear
void insertRear(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    n->next = head;
    n->prev = head->prev;
    head->prev->next = n;
    head->prev = n;
}

// 🟣 Insert by Position
void insertPos(int val, int pos) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    struct Node *t = head->next;
    int i = 1;

    while (t != head && i < pos) {
        t = t->next;
        i++;
    }
    if (t == head && i < pos) {
        printf("Invalid Position!\n");
        free(n);
        return;
    }

    n->next = t;
    n->prev = t->prev;
    t->prev->next = n;
    t->prev = n;
}

// 🔴 Delete Front
void deleteFront() {
    if (head->next == head) {
        printf("List Empty!\n");
        return;
    }
    struct Node *t = head->next;
    head->next = t->next;
    t->next->prev = head;
    free(t);
}

// 🟠 Delete Rear
void deleteRear() {
    if (head->next == head) {
        printf("List Empty!\n");
        return;
    }
    struct Node *t = head->prev;
    t->prev->next = head;
    head->prev = t->prev;
    free(t);
}

// 🟡 Delete by Key
void deleteKey(int key) {
    struct Node *t = head->next;
    while (t != head && t->data != key)
        t = t->next;
    if (t == head) {
        printf("Key not found!\n");
        return;
    }
    t->prev->next = t->next;
    t->next->prev = t->prev;
    free(t);
}

// 🔍 Search by Key
void searchKey(int key) {
    struct Node *t = head->next;
    int pos = 1;
    while (t != head) {
        if (t->data == key) {
            printf("Key %d found at position %d\n", key, pos);
            return;
        }
        t = t->next;
        pos++;
    }
    printf("Key not found!\n");
}

// 🧮 Ordered Insert
void insertOrdered(int val) {
    struct Node *n = (struct Node*)malloc(sizeof(struct Node));
    n->data = val;
    struct Node *t = head->next;

    while (t != head && t->data < val)
        t = t->next;

    n->next = t;
    n->prev = t->prev;
    t->prev->next = n;
    t->prev = n;
}

// 🔁 Reverse List
void reverseList() {
    struct Node *cur = head;
    struct Node *temp = NULL;

    do {
        temp = cur->next;
        cur->next = cur->prev;
        cur->prev = temp;
        cur = temp;
    } while (cur != head);

    printf("List Reversed!\n");
}
```

```c
// 📋 Copy List
struct Node* copyList() {
    struct Node *copyHead = (struct Node*)malloc(sizeof(struct Node));
    copyHead->data = -1;
    copyHead->next = copyHead;
    copyHead->prev = copyHead;

    struct Node *src = head->next;
    while (src != head) {
        struct Node *n = (struct Node*)malloc(sizeof(struct Node));
        n->data = src->data;
        n->next = copyHead;
        n->prev = copyHead->prev;
        copyHead->prev->next = n;
        copyHead->prev = n;
        src = src->next;
    }
    printf("List Copied!\n");
    return copyHead;
}

// 👁️ Display
void display(struct Node *h) {
    if (h->next == h) {
        printf("List Empty!\n");
        return;
    }
    struct Node *t = h->next;
    while (t != h) {
        printf("%d <-> ", t->data);
        t = t->next;
    }
    printf("(Back to Head)\n");
}

// 🧠 Main Function
int main() {
    int ch, val, pos, key;
    struct Node *copy = NULL;
    createHeader();

    do {
        printf("\n======= Circular Doubly Linked List Menu =======\n");
        printf("1.Insert Front   2.Insert Rear   3.Insert Position\n");
        printf("4.Delete Front   5.Delete Rear   6.Delete Key\n");
        printf("7.Search Key     8.Ordered Insert   9.Reverse List\n");
        printf("10.Copy List     11.Display        12.Exit\n");
        printf("================================================\n");
        printf("Enter choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1: printf("Value: "); scanf("%d", &val); insertFront(val);
break;
            case 2: printf("Value: "); scanf("%d", &val); insertRear(val);
break;
            case 3: printf("Value & Position: "); scanf("%d%d", &val, &pos);
insertPos(val, pos); break;
            case 4: deleteFront(); break;
            case 5: deleteRear(); break;
            case 6: printf("Key: "); scanf("%d", &key); deleteKey(key); break;
            case 7: printf("Key: "); scanf("%d", &key); searchKey(key); break;
            case 8: printf("Value: "); scanf("%d", &val); insertOrdered(val);
break;
            case 9: reverseList(); break;
            case 10: copy = copyList(); display(copy); break;
            case 11: display(head); break;
            case 12: printf("Exiting...\n"); break;
            default: printf("Invalid Choice!\n");
        }
    } while (ch != 12);

    return 0;
}
```

Infix → Postfix Conversion and Evaluation 📘

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

// 🧮 Stack for operators
char stack[50];
int top = -1;

// 🟢 Push operation
void push(char c) {
    stack[++top] = c;
}

// 🔴 Pop operation
char pop() {
    return stack[top--];
```

```c
}

// ⚖️ Precedence function
int prec(char c) {
    if (c == '^') return 3;
    if (c == '*' || c == '/') return 2;
    if (c == '+' || c == '-') return 1;
    return 0;
}

// 🧩 Convert Infix to Postfix
void infixToPostfix(char inf[], char post[]) {
    int i, k = 0;
    char ch;

    for (i = 0; inf[i]; i++) {
        ch = inf[i];

        // Operand ➜ directly add to postfix
        if (isalnum(ch))
            post[k++] = ch;

        // '(' ➜ push to stack
        else if (ch == '(')
            push(ch);

        // ')' ➜ pop till '('
        else if (ch == ')') {
            while (top != -1 && stack[top] != '(')
                post[k++] = pop();
            top--; // remove '('
        }

        // Operator ➜ manage precedence
        else {
            while (top != -1 && prec(stack[top]) >= prec(ch))
                post[k++] = pop();
            push(ch);
        }
    }

    // Pop remaining operators
    while (top != -1)
        post[k++] = pop();

    post[k] = '\0';
}
```

```c
// 🧠 Evaluate Postfix Expression
int evalPostfix(char post[]) {
    int s[50], t = -1, a, b;

    for (int i = 0; post[i]; i++) {
        if (isdigit(post[i]))
            s[++t] = post[i] - '0'; // convert char to int
        else {
            b = s[t--];
            a = s[t--];
            switch (post[i]) {
                case '+': s[++t] = a + b; break;
                case '-': s[++t] = a - b; break;
                case '*': s[++t] = a * b; break;
                case '/': s[++t] = a / b; break;
                case '^': {
                    int res = 1;
                    for (int j = 0; j < b; j++) res *= a;
                    s[++t] = res;
                    break;
                }
            }
        }
    }
    return s[t];
}

// 🏁 Main Function
int main() {
    char inf[50], post[50];
    printf("Enter Infix Expression: ");
    scanf("%s", inf);

    infixToPostfix(inf, post);

    printf("\nPostfix Expression: %s\n", post);
    printf("Evaluated Result: %d\n", evalPostfix(post));

    return 0;
}
```

📘 Infix → Prefix Conversion and Evaluation

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

```c
char stack[50];
int top = -1;

// 🟢 Push element to stack
void push(char c) {
    stack[++top] = c;
}

// 🔴 Pop element from stack
char pop() {
    return stack[top--];
}

// ⚖️ Operator precedence
int prec(char c) {
    if (c == '^') return 3;
    if (c == '*' || c == '/') return 2;
    if (c == '+' || c == '-') return 1;
    return 0;
}

// 🔄 Reverse a string
void reverse(char *s) {
    int i, j;
    char t;
    for (i = 0, j = strlen(s) - 1; i < j; i++, j--) {
        t = s[i];
        s[i] = s[j];
        s[j] = t;
    }
}

// 🧩 Convert Infix → Prefix
void infixToPrefix(char inf[], char pre[]) {
    reverse(inf); // Step 1: Reverse infix expression

    // Step 2: Swap '(' with ')'
    for (int i = 0; inf[i]; i++) {
        if (inf[i] == '(')
            inf[i] = ')';
        else if (inf[i] == ')')
            inf[i] = '(';
    }

    char post[50];
    int k = 0;
    char ch;
    // Step 3: Convert reversed infix to postfix
    for (int i = 0; inf[i]; i++) {
        ch = inf[i];

        if (isalnum(ch))
            post[k++] = ch;
        else if (ch == '(')
            push(ch);
        else if (ch == ')') {
            while (top != -1 && stack[top] != '(')
                post[k++] = pop();
            top--;
        } else {
            while (top != -1 && prec(stack[top]) >= prec(ch))
                post[k++] = pop();
            push(ch);
        }
    }

    // Step 4: Pop remaining operators
    while (top != -1)
        post[k++] = pop();

    post[k] = '\0';

    // Step 5: Reverse postfix → get prefix
    reverse(post);
    strcpy(pre, post);
}

// 🧠 Evaluate Prefix Expression
int evalPrefix(char pre[]) {
    int s[50], t = -1, a, b;

    // Start scanning from right to left
    for (int i = strlen(pre) - 1; i >= 0; i--) {
        if (isdigit(pre[i]))
            s[++t] = pre[i] - '0';
        else {
            a = s[t--];
            b = s[t--];
            switch (pre[i]) {
                case '+': s[++t] = a + b; break;
                case '-': s[++t] = a - b; break;
                case '*': s[++t] = a * b; break;
                case '/': s[++t] = a / b; break;
                case '^': {
                    int res = 1;
                for (int j = 0; j < b; j++) res *= a;
                s[++t] = res;
                break;
            }
        }
    }
}

    return s[t];
}

// 🏁 Main function
int main() {
    char inf[50], pre[50];
    printf("Enter Infix Expression: ");
    scanf("%s", inf);

    infixToPrefix(inf, pre);

    printf("\nPrefix Expression: %s\n", pre);
    printf("Evaluated Result: %d\n", evalPrefix(pre));

    return 0;
}
```

📘 Ordinary Queue (Static + Dynamic Implementation)

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

#define MAX 50

// 🧩 Structure Definition
struct Queue {
    int *q;
    int front, rear, size;
};

// 🏗️ Initialize the Queue (Dynamic Memory Allocation)
void initStatic(struct Queue *Q, int n) {
    Q->q = (int*)malloc(n * sizeof(int));
    Q->size = n;
    Q->front = 0;
    Q->rear = -1;
}
```

```c
// 🔍 Check if Queue is Full
int isFull(struct Queue *Q) {
    return Q->rear == Q->size - 1;
}

// 🔍 Check if Queue is Empty
int isEmpty(struct Queue *Q) {
    return Q->front > Q->rear;
}

// 🟢 Enqueue Operation (Insert Element)
void enqueue(struct Queue *Q, int val) {
    if (isFull(Q))
        printf("⚠️ Queue is Full!\n");
    else
        Q->q[++Q->rear] = val;
}

// 🔴 Dequeue Operation (Delete Element)
void dequeue(struct Queue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue is Empty!\n");
    else
        printf("🗑️ Deleted: %d\n", Q->q[Q->front++]);
}

// 👀 Display Queue Elements
void display(struct Queue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue is Empty!\n");
    else {
        printf("📋 Queue Elements: ");
        for (int i = Q->front; i <= Q->rear; i++)
            printf("%d ", Q->q[i]);
        printf("\n");
    }
}

// 🏁 Main Function
int main() {
    struct Queue Q;
    int ch, val, n;

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

    initStatic(&Q, n);
    do {
        printf("\n🔹 MENU 🔹\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                enqueue(&Q, val);
                break;

            case 2:
                dequeue(&Q);
                break;

            case 3:
                display(&Q);
                break;

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

            default:
                printf("⚠️ Invalid Choice! Try again.\n");
        }

    } while (ch != 4);

    free(Q.q); // 🧹 Free dynamically allocated memory
    return 0;
}
```

circular Queue (Static + Dynamic)

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

// 🧩 Structure Definition
struct CQueue {
    int *cq;
```

```c
    int front, rear, size;
};

// ⚙️ Initialize the Circular Queue
void initCQ(struct CQueue *Q, int n) {
    Q->cq = (int*)malloc(n * sizeof(int));
    Q->size = n;
    Q->front = Q->rear = -1;
}

// 🔍 Check if Queue is Full
int isFull(struct CQueue *Q) {
    return (Q->rear + 1) % Q->size == Q->front;
}

// 🔍 Check if Queue is Empty
int isEmpty(struct CQueue *Q) {
    return Q->front == -1;
}

// 🟢 Enqueue Operation (Insert Element)
void enqueue(struct CQueue *Q, int val) {
    if (isFull(Q))
        printf("⚠️ Queue Full!\n");
    else {
        if (Q->front == -1)
            Q->front = 0;
        Q->rear = (Q->rear + 1) % Q->size;
        Q->cq[Q->rear] = val;
        printf("✅ Inserted: %d\n", val);
    }
}

// 🔴 Dequeue Operation (Delete Element)
void dequeue(struct CQueue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue Empty!\n");
    else {
        printf("🗑️ Deleted: %d\n", Q->cq[Q->front]);
        if (Q->front == Q->rear)
            Q->front = Q->rear = -1;
        else
            Q->front = (Q->front + 1) % Q->size;
    }
}

// 👀 Display Queue Elements
void display(struct CQueue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue Empty!\n");
    else {
        int i = Q->front;
        printf("📋 Queue: ");
        while (1) {
            printf("%d ", Q->cq[i]);
            if (i == Q->rear) break;
            i = (i + 1) % Q->size;
        }
        printf("\n");
    }
}

// 🏁 Main Function
int main() {
    struct CQueue Q;
    int ch, val, n;

    printf("Enter size of Circular Queue: ");
    scanf("%d", &n);

    initCQ(&Q, n);

    do {
        printf("\n🔹 MENU 🔹\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                enqueue(&Q, val);
                break;
            case 2:
                dequeue(&Q);
                break;
            case 3:
                display(&Q);
                break;
            case 4:
                printf("👋 Exiting...\n");
                break;
            default:
                printf("⚠️ Invalid Choice! Try again.\n");
        }
    } while (ch != 4);

    free(Q.cq); // 🧹 Free allocated memory
    return 0;
}
```

Double Ended Queue (DEQueue)

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

// 🧩 Structure Definition
struct DEQueue {
    int *dq;
    int front, rear, size;
};

// ⚙️ Initialize DEQueue
void init(struct DEQueue *Q, int n) {
    Q->dq = (int*)malloc(n * sizeof(int));
    Q->size = n;
    Q->front = -1;
    Q->rear = -1;
}

// 🔍 Check if Queue is Full
int isFull(struct DEQueue *Q) {
    return (Q->front == 0 && Q->rear == Q->size - 1) || (Q->front == Q->rear +
1);
}

// 🔍 Check if Queue is Empty
int isEmpty(struct DEQueue *Q) {
    return Q->front == -1;
}

// 🟢 Insert at Front
void insertFront(struct DEQueue *Q, int val) {
    if (isFull(Q))
        printf("⚠️ Queue Full!\n");
    else {
        if (Q->front == -1)
            Q->front = Q->rear = 0;
        else if (Q->front == 0)
```

```c
            Q->front = Q->size - 1;
        else
            Q->front--;
        Q->dq[Q->front] = val;
        printf("✅ Inserted (Front): %d\n", val);
    }
}

// 🟢 Insert at Rear
void insertRear(struct DEQueue *Q, int val) {
    if (isFull(Q))
        printf("⚠️ Queue Full!\n");
    else {
        if (Q->front == -1)
            Q->front = Q->rear = 0;
        else if (Q->rear == Q->size - 1)
            Q->rear = 0;
        else
            Q->rear++;
        Q->dq[Q->rear] = val;
        printf("✅ Inserted (Rear): %d\n", val);
    }
}

// 🔴 Delete from Front
void deleteFront(struct DEQueue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue Empty!\n");
    else {
        printf("🗑️ Deleted (Front): %d\n", Q->dq[Q->front]);
        if (Q->front == Q->rear)
            Q->front = Q->rear = -1;
        else if (Q->front == Q->size - 1)
            Q->front = 0;
        else
            Q->front++;
    }
}

// 🔴 Delete from Rear
void deleteRear(struct DEQueue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue Empty!\n");
    else {
        printf("🗑️ Deleted (Rear): %d\n", Q->dq[Q->rear]);
        if (Q->front == Q->rear)
            Q->front = Q->rear = -1;
        else if (Q->rear == 0)
```

```c
            Q->rear = Q->size - 1;
        else
            Q->rear--;
    }
}

// 👀 Display DEQueue
void display(struct DEQueue *Q) {
    if (isEmpty(Q))
        printf("⚠️ Queue Empty!\n");
    else {
        int i = Q->front;
        printf("📋 Queue: ");
        while (1) {
            printf("%d ", Q->dq[i]);
            if (i == Q->rear)
                break;
            i = (i + 1) % Q->size;
        }
        printf("\n");
    }
}

// 🏁 Main Function
int main() {
    struct DEQueue Q;
    int ch, val, n;

    printf("Enter size of DEQueue: ");
    scanf("%d", &n);

    init(&Q, n);

    do {
        printf("\n🔹 MENU 🔹\n");
        printf("1. Insert Front\n");
        printf("2. Insert Rear\n");
        printf("3. Delete Front\n");
        printf("4. Delete Rear\n");
        printf("5. Display\n");
        printf("6. Exit\n");
        printf("Choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter Value: ");
                scanf("%d", &val);
                insertFront(&Q, val);
                break;
            case 2:
                printf("Enter Value: ");
                scanf("%d", &val);
                insertRear(&Q, val);
                break;
            case 3:
                deleteFront(&Q);
                break;
            case 4:
                deleteRear(&Q);
                break;
            case 5:
                display(&Q);
                break;
            case 6:
                printf("👋 Exiting...\n");
                break;
            default:
                printf("⚠️ Invalid Choice!\n");
        }
    } while (ch != 6);

    free(Q.dq); // 🧹 Free allocated memory
    return 0;
}
```

priority Queue (Static + Dynamic)

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

// 🧩 Structure for Node (holds data and priority)
struct Node {
    int data, pr;
};

// 🧩 Structure for Priority Queue
struct PQueue {
    struct Node *pq;
    int size, count;
};

// ⚙️ Initialize Priority Queue
void initPQ(struct PQueue *Q, int n) {
    Q->pq = (struct Node*)malloc(n * sizeof(struct Node));
```

```c
    Q->size = n;
    Q->count = 0;
}

// 🟢 Insert element based on priority
void insert(struct PQueue *Q, int val, int p) {
    if (Q->count == Q->size)
        printf("⚠️ Queue Full!\n");
    else {
        int i = Q->count - 1;
        // Shift elements based on priority (lower value = higher priority)
        while (i >= 0 && Q->pq[i].pr > p) {
            Q->pq[i + 1] = Q->pq[i];
            i--;
        }
        Q->pq[i + 1].data = val;
        Q->pq[i + 1].pr = p;
        Q->count++;
        printf("✅ Inserted: %d (Priority: %d)\n", val, p);
    }
}

// 🔴 Delete element with highest priority (lowest 'pr' value)
void delete(struct PQueue *Q) {
    if (Q->count == 0)
        printf("⚠️ Queue Empty!\n");
    else {
        printf("🗑️ Deleted: %d (Priority %d)\n", Q->pq[0].data, Q->pq[0].pr);
        for (int i = 1; i < Q->count; i++)
            Q->pq[i - 1] = Q->pq[i];
        Q->count--;
    }
}

// 👀 Display Priority Queue
void display(struct PQueue *Q) {
    if (Q->count == 0)
        printf("⚠️ Queue Empty!\n");
    else {
        printf("📋 Priority Queue:\n");
        for (int i = 0; i < Q->count; i++)
            printf("%d(%d) ", Q->pq[i].data, Q->pq[i].pr);
        printf("\n");
    }
}

// 🏁 Main Function
int main() {
    struct PQueue Q;
    int ch, val, pr, n;

    printf("Enter size of Priority Queue: ");
    scanf("%d", &n);

    initPQ(&Q, n);

    do {
        printf("\n🔹 MENU 🔹\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value & priority: ");
                scanf("%d%d", &val, &pr);
                insert(&Q, val, pr);
                break;
            case 2:
                delete(&Q);
                break;
            case 3:
                display(&Q);
                break;
            case 4:
                printf("👋 Exiting...\n");
                break;
            default:
                printf("⚠️ Invalid Choice!\n");
        }
    } while (ch != 4);

    free(Q.pq); // 🧹 Free dynamically allocated memory
    return 0;
}
```