LINKED LIST:-

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

// Node structure
typedef struct node {
    int data;
    struct node* link;
} Node;

// Global start pointer
Node* start = NULL;

// Function prototypes
Node* createNode(int data);
void display();
void insertAtBeginning(int x);
void insertAtEnd(int x);
void insertAtPosition(int x, int pos);
void deleteAtBeginning();
void deleteAtEnd();
void deleteAtPosition(int pos);
void search(int x);
int countNodes();
Node* copyList();
void displayCopied(Node* copiedStart);

// Helper: Create new node
Node* createNode(int data) {
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->data = data;
    tmp->link = NULL;
    return tmp;
}

// Display list
void display() {
    if(start == NULL) {
```

```
        printf("Empty list\n");
        return;
    }

    Node* ptr = start;
    while(ptr != NULL) {
        printf("%d -> ", ptr->data);
        ptr = ptr->link;
    }
    printf("NULL\n");
}

// Insert at beginning
void insertAtBeginning(int x) {
    Node* tmp = createNode(x);
    tmp->link = start;
    start = tmp;
    printf("Inserted %d at beginning\n", x);
}

// Insert at end
void insertAtEnd(int x) {
    Node* tmp = createNode(x);

    if(start == NULL) {
        start = tmp;
    } else {
        Node* ptr = start;
        while(ptr->link != NULL) {
            ptr = ptr->link;
        }
        ptr->link = tmp;
    }
    printf("Inserted %d at end\n", x);
}

// Insert at position (following your handwritten algorithm)
void insertAtPosition(int x, int pos) {
```

```
    Node *ptr, *tmp;
    tmp = createNode(x);

    // Handle position 1 (insert at beginning)
    if(pos == 1) {
        tmp->link = start;
        start = tmp;
        printf("Inserted %d at position %d\n", x, pos);
        return;
    }

    ptr = start;

    // Move to (pos-1)th node WITH NULL CHECK
    for(int i = 1; i < pos-1 && ptr != NULL; i++) {
        ptr = ptr->link;
    }

    // Validate position
    if(ptr == NULL) {
        printf("Invalid position\n");
        free(tmp);  // Don't leak memory!
        return;
    }

    tmp->link = ptr->link;
    ptr->link = tmp;
    printf("Inserted %d at position %d\n", x, pos);
}

// Delete at beginning
void deleteAtBeginning() {
    if(start == NULL) {
        printf("List is empty\n");
        return;
    }

    Node* tmp = start;
```

```
    start = start->link;
    printf("Deleted %d from beginning\n", tmp->data);
    free(tmp);
}

// Delete at end
void deleteAtEnd() {
    if(start == NULL) {
        printf("List is empty\n");
        return;
    }

    if(start->link == NULL) {
        printf("Deleted %d from end\n", start->data);
        free(start);
        start = NULL;
        return;
    }

    Node* ptr = start;
    while(ptr->link->link != NULL) {
        ptr = ptr->link;
    }

    Node* tmp = ptr->link;
    ptr->link = NULL;
    printf("Deleted %d from end\n", tmp->data);
    free(tmp);
}

//delete at position
void deleteAtPosition(int pos) {
    Node *ptr, *tmp;

    if(start == NULL) {
        printf("List is empty\n");
        return;
    }
```

```
    if(pos == 1) {
        tmp = start;
        start = start->link;
    } else {
        ptr = start;
        for(int i = 1; i < pos-1 && ptr != NULL; i++)
            ptr = ptr->link;

        if(ptr == NULL || ptr->link == NULL) {
            printf("Invalid position\n");
            return;
        }
        tmp = ptr->link;
        ptr->link = tmp->link;
    }

    printf("Deleted %d from position %d\n", tmp->data, pos);
    free(tmp);
}

// Search element
void search(int x) {
    Node* ptr = start;
    int pos = 1;

    while(ptr != NULL) {
        if(ptr->data == x) {
            printf("Found at position %d\n", pos);
            return;
        }
        ptr = ptr->link;
        pos++;
    }
    printf("Not found\n");
}

// Count nodes
```

```c
int countNodes() {
    int c = 0;
    Node* ptr = start;

    while(ptr != NULL) {
        c++;
        ptr = ptr->link;
    }

    printf("Total nodes: %d\n", c);
    return c;
}

//merge list
Node* mergeLists(Node* l1, Node* l2) {
    Node *merged = NULL, *ptr = NULL;

    while(l1 != NULL && l2 != NULL) {
        if(l1->data < l2->data) {
            if(merged == NULL)
                merged = ptr = createNode(l1->data);
            else {
                ptr->link = createNode(l1->data);
                ptr = ptr->link;
            }
            l1 = l1->link;
        } else {
            if(merged == NULL)
                merged = ptr = createNode(l2->data);
            else {
                ptr->link = createNode(l2->data);
                ptr = ptr->link;
            }
            l2 = l2->link;
        }
    }

    while(l1 != NULL) {
```

```
        ptr->link = createNode(l1->data);
        ptr = ptr->link;
        l1 = l1->link;
    }

    while(l2 != NULL) {
        ptr->link = createNode(l2->data);
        ptr = ptr->link;
        l2 = l2->link;
    }

    return merged;
}

// Copy list
Node* copyList() {
    if(start == NULL) {
        printf("Original list is empty\n");
        return NULL;
    }

    Node *newStart = NULL, *newPtr = NULL, *ptr = start;

    while(ptr != NULL) {
        Node* tmp = createNode(ptr->data);

        if(newStart == NULL) {
            newStart = tmp;
            newPtr = tmp;
        } else {
            newPtr->link = tmp;
            newPtr = tmp;
        }
        ptr = ptr->link;
    }

    printf("List copied successfully\n");
    return newStart;
```

```c
}

// Display copied list
void displayCopied(Node* copiedStart) {
    if(copiedStart == NULL) {
        printf("Copied list is empty\n");
        return;
    }

    printf("Copied List: ");
    Node* ptr = copiedStart;

    while(ptr != NULL) {
        printf("%d", ptr->data);
        if(ptr->link != NULL) printf(" -> ");
        ptr = ptr->link;
    }
    printf(" -> NULL\n");
}

// Main function
int main() {
    int ch, val, pos;
    Node* copiedStart = NULL;

    printf("=== LINKED LIST OPERATIONS ===\n");

    while(1) {
        printf("\n1.Display  2.Insert-Beg  3.Insert-End  4.Insert-Pos\n");
        printf("5.Delete-Beg  6.Delete-End  7.Delete-Pos  8.Search\n");
        printf("9.Count  10.Copy  11.Display-Copy 12.merge  13.Exit\n");
        printf("Choice: ");

        scanf("%d", &ch);

        switch(ch) {
            case 1:
                display();
```

```
            break;
        case 2:
            printf("Value: ");
            scanf("%d", &val);
            insertAtBeginning(val);
            break;
        case 3:
            printf("Value: ");
            scanf("%d", &val);
            insertAtEnd(val);
            break;
        case 4:
            printf("Value: ");
            scanf("%d", &val);
            printf("Position: ");
            scanf("%d", &pos);
            insertAtPosition(val, pos);
            break;
        case 5:
            deleteAtBeginning();
            break;
        case 6:
            deleteAtEnd();
            break;
        case 7:
            printf("Position: ");
            scanf("%d", &pos);
            deleteAtPosition(pos);
            break;
        case 8:
            printf("Search value: ");
            scanf("%d", &val);
            search(val);
            break;
        case 9:
            countNodes();
            break;
        case 10:
```

```
                copiedStart = copyList();
                break;
            case 11:
                displayCopied(copiedStart);
                break;
            case 12:
                Node* merged = mergeLists(start, copiedStart);
                displayCopied(merged);
            case 13:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Doubly Linked List:-

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

struct node {
    int data;
    struct node *prev, *next;
};

struct node *start = NULL;

/* Create node */
struct node* create(int x) {
    struct node *tmp = (struct node*)malloc(sizeof(struct node));
    tmp->data = x;
    tmp->prev = tmp->next = NULL;
    return tmp;
}

/* Display */
void display() {
    struct node *ptr = start;
    if (start == NULL) {
        printf("Empty list\n");
        return;
    }
    while (ptr != NULL) {
        printf("%d <-> ", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}
```

```c
/* Insert at beginning */
void insertBeg(int x) {
    struct node *tmp = create(x);
    if (start != NULL)
        start->prev = tmp;
    tmp->next = start;
    start = tmp;
}

/* Insert at end */
void insertEnd(int x) {
    struct node *tmp = create(x);
    if (start == NULL) {
        start = tmp;
        return;
    }
    struct node *ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = tmp;
    tmp->prev = ptr;
}

/* Insert at position */
void insertPos(int x, int pos) {
    struct node *ptr = start;

    if (pos == 1) {
        insertBeg(x);
        return;
    }
```

```
    for (int i = 1; i < pos - 1 && ptr != NULL; i++)
        ptr = ptr->next;

    if (ptr == NULL) {
        printf("Invalid position\n");
        return;
    }

    struct node *tmp = create(x);
    tmp->next = ptr->next;
    tmp->prev = ptr;

    if (ptr->next != NULL)
        ptr->next->prev = tmp;

    ptr->next = tmp;
}

/* Delete at beginning */
void deleteBeg() {
    if (start == NULL) return;

    struct node *tmp = start;
    start = start->next;

    if (start != NULL)
        start->prev = NULL;

    free(tmp);
}

/* Delete at end */
void deleteEnd() {
```

```c
    if (start == NULL) return;

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

    struct node *ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;

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

/* Delete at position */
void deletePos(int pos) {
    struct node *ptr = start;

    if (start == NULL) return;

    if (pos == 1) {
        deleteBeg();
        return;
    }

    for (int i = 1; i < pos && ptr != NULL; i++)
        ptr = ptr->next;

    if (ptr == NULL) {
        printf("Invalid position\n");
        return;
```

```
    }

    ptr->prev->next = ptr->next;
    if (ptr->next != NULL)
        ptr->next->prev = ptr->prev;

    free(ptr);
}

/* Main */
int main() {
    insertBeg(10);
    insertEnd(20);
    insertEnd(40);
    insertPos(30, 3);

    display();

    deleteBeg();
    display();

    deleteEnd();
    display();

    deletePos(2);
    display();

    return 0;
}
```

Circular linked list :-

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

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

Node *last = NULL;

/* Create node */
Node* createNode(int x) {
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->data = x;
    tmp->next = tmp;
    return tmp;
}

/* Display */
void display() {
    Node *ptr;
    if(last == NULL) {
        printf("Empty list\n");
        return;
    }
    ptr = last->next;
    do {
        printf("%d -> ", ptr->data);
        ptr = ptr->next;
    } while(ptr != last->next);
    printf("(back to start)\n");
}

/* Insert at beginning */
void insertBeg(int x) {
    Node *tmp = createNode(x);
```

```
    if(last == NULL) {
        last = tmp;
        return;
    }
    tmp->next = last->next;
    last->next = tmp;
}

/* Insert at end */
void insertEnd(int x) {
    Node *tmp = createNode(x);
    if(last == NULL) {
        last = tmp;
        return;
    }
    tmp->next = last->next;
    last->next = tmp;
    last = tmp;
}

/* Insert at position */
void insertAtPos(int x, int pos) {
    Node *ptr = last->next, *tmp;
    if(pos == 1) {
        insertBeg(x);
        return;
    }
    for(int i = 1; i < pos-1 && ptr->next != last->next; i++)
        ptr = ptr->next;
    tmp = createNode(x);
    tmp->next = ptr->next;
    ptr->next = tmp;
    if(ptr == last)
        last = tmp;
}

/* Delete at beginning */
void deleteBeg() {
```

```
    Node *tmp;
    if(last == NULL) return;
    tmp = last->next;
    if(tmp == last) {
        free(last);
        last = NULL;
        return;
    }
    last->next = tmp->next;
    free(tmp);
}

/* Delete at end */
void deleteEnd() {
    Node *ptr;
    if(last == NULL) return;
    if(last->next == last) {
        free(last);
        last = NULL;
        return;
    }
    ptr = last->next;
    while(ptr->next != last)
        ptr = ptr->next;
    ptr->next = last->next;
    free(last);
    last = ptr;
}

/* Delete at position */
void deleteAtPos(int pos) {
    Node *ptr = last->next, *tmp;
    if(last == NULL) return;
    if(pos == 1) {
        deleteBeg();
        return;
    }
    for(int i = 1; i < pos-1 && ptr->next != last->next; i++)
```

```
        ptr = ptr->next;
    tmp = ptr->next;
    ptr->next = tmp->next;
    if(tmp == last)
        last = ptr;
    free(tmp);
}

int main() {
    insertBeg(10);
    insertEnd(20);
    insertEnd(40);
    insertAtPos(30, 3);
    display();
    deleteAtPos(2);
    display();
    return 0;
}
```

POLYNOMIAL ADDITION:-

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

typedef struct node {
    int c, p;
    struct node *next;
} NODE;

/* Insert at end */
NODE* insert(NODE* head, int c, int p) {
    NODE* n = (NODE*)malloc(sizeof(NODE));
    n->c = c;
    n->p = p;
    n->next = NULL;

    if (head == NULL)
        return n;

    NODE* t = head;
    while (t->next != NULL)
        t = t->next;

    t->next = n;
    return head;
}

/* Add polynomials */
NODE* add(NODE* p1, NODE* p2) {
    NODE* res = NULL;

    while (p1 && p2) {
        if (p1->p == p2->p) {
            res = insert(res, p1->c + p2->c, p1->p);
            p1 = p1->next;
            p2 = p2->next;
        }
        else if (p1->p > p2->p) {
```

```
            res = insert(res, p1->c, p1->p);
            p1 = p1->next;
        }
        else {
            res = insert(res, p2->c, p2->p);
            p2 = p2->next;
        }
    }

    while (p1) {
        res = insert(res, p1->c, p1->p);
        p1 = p1->next;
    }

    while (p2) {
        res = insert(res, p2->c, p2->p);
        p2 = p2->next;
    }

    return res;
}

/* Display */
void display(NODE* t) {
    while (t) {
        printf("%dx^%d", t->c, t->p);
        t = t->next;
        if (t) printf(" + ");
    }
    printf("\n");
}

int main() {
    NODE *p1 = NULL, *p2 = NULL, *sum;

    p1 = insert(p1, 3, 3);
    p1 = insert(p1, 2, 2);
    p1 = insert(p1, 5, 1);
```

```
    p2 = insert(p2, 4, 3);
    p2 = insert(p2, 1, 2);
    p2 = insert(p2, 2, 0);

    sum = add(p1, p2);

    display(sum);
    return 0;
}
```

Analysis :-

Structure	Insert	Delete	Traverse	Space
Array List	O(n)	O(n)	O(n)	Fixed
Singly LL	O(1)/O(n)	O(1)/O(n)	O(n)	Dynamic
Circular LL	O(1)	O(n)	O(n)	Dynamic
Doubly LL	O(1)	O(1)	O(n)	More

STACK (ARRAY IMPLEMENTATION):-

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

#define MAX 100

// Global variables (no structure needed)
int stack[MAX];
int top = -1;

// Function prototypes
void initStack();
int isEmpty();
int isFull();
void push(int data);
int pop();
int peek();
void display();

// Initialize stack
void initStack() {
    top = -1;
    printf("Stack initialized\n");
}

// Check if stack is empty
int isEmpty() {
    return (top == -1);
}

// Check if stack is full
int isFull() {
    return (top == MAX - 1);
}

// Push operation
void push(int data) {
    if (isFull()) {
        printf("Stack Overflow! Cannot push %d\n", data);
        return;
    }
```

```
    top++;
    stack[top] = data;
    printf("Pushed %d onto stack\n", data);
}

// Pop operation
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    }

    int data = stack[top];
    top--;
    printf("Popped %d from stack\n", data);
    return data;
}

// Peek operation (view top element)
int peek() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return -1;
    }

    printf("Top element: %d\n", stack[top]);
    return stack[top];
}

// Display stack
void display() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }

    printf("Stack elements (top to bottom): ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
```

```c
}

// Main function
int main() {
    int choice, data;

    printf("=== STACK IMPLEMENTATION USING ARRAY ===\n");
    initStack();

    while (1) {
        printf("\n--- Menu ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek/Top\n");
        printf("4. Display\n");
        printf("5. Check if Empty\n");
        printf("6. Check if Full\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to push: ");
                scanf("%d", &data);
                push(data);
                break;
            case 2:
                pop();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
                if (isEmpty()) {
                    printf("Stack is empty\n");
                } else {
                    printf("Stack is not empty\n");
```

```
                }
                break;
            case 6:
                if (isFull()) {
                    printf("Stack is full\n");
                } else {
                    printf("Stack is not full\n");
                }
                break;
            case 7:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

STACK (LINKEDLIST IMPLEMENTATION):-

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

// Node structure
typedef struct node {
    int data;
    struct node* next;
} Node;

// Global top pointer
Node* top = NULL;

// Function prototypes
Node* createNode(int data);
int isEmpty();
void push(int data);
int pop();
int peek();
void display();

// Helper: Create new node
Node* createNode(int data) {
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->data = data;
    tmp->next = NULL;
    return tmp;
}

// Check if empty
int isEmpty() {
    return (top == NULL);
}

// Push (insert at top)
void push(int data) {
    Node* tmp = createNode(data);
    tmp->next = top;
    top = tmp;
    printf("Pushed %d\n", data);
}
```

```c
// Pop (delete from top)
int pop() {
    if(isEmpty()) {
        printf("Stack Underflow\n");
        return -1;
    }

    Node* tmp = top;
    int data = tmp->data;
    top = top->next;
    free(tmp);
    printf("Popped %d\n", data);
    return data;
}

// Peek (view top)
int peek() {
    if(isEmpty()) {
        printf("Stack empty\n");
        return -1;
    }

    printf("Top: %d\n", top->data);
    return top->data;
}

// Display stack
void display() {
    if(isEmpty()) {
        printf("Stack empty\n");
        return;
    }

    printf("Stack: ");
    Node* ptr = top;
    while(ptr != NULL) {
        printf("%d\t", ptr->data);
        ptr = ptr->next;
    }
}
```

```c
// Main function
int main() {
    int ch, val;

    printf("=== STACK (LINKED LIST) ===\n");

    while(1) {
        printf("\n1.Push  2.Pop  3.Peek  4.Display  5.Empty?  6.Exit\n");
        printf("Choice: ");
        scanf("%d", &ch);

        switch(ch) {
            case 1:
                printf("Value: ");
                scanf("%d", &val);
                push(val);
                break;
            case 2:
                pop();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
                printf(isEmpty() ? "Empty\n" : "Not Empty\n");
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid!\n");
        }
    }
    return 0;
}
```

QUEUE (ARRAY IMPLEMENTATION):-

```c
#include <stdio.h>
#define MAX 5

int arr[MAX];
int front = -1, rear = -1;

// Enqueue
void enqueue(int x) {
    if (rear == MAX - 1) {
        printf("Queue Full\n");
        return;
    }

    if (front == -1 && rear==-1){
        front =rear= 0;
    }else{
        rear++;
    }
    arr[rear] = x;
}

// Dequeue
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Empty\n");
        return;
    }

    printf("Removed %d\n", arr[front]);
```

```c
    front++;

    // Reset when queue becomes empty
    if (front > rear)
        front = rear = -1;
}

// Display
void display() {
    if (front == -1) {
        printf("Queue Empty\n");
        return;
    }

    for (int i = front; i <= rear; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int ch, x;

    while (1) {
        printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
```

```
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

QUEUE (LINKEDLIST IMPLEMENTATION):-

IN NOTEBOOK

INFIXTOPOSTFIX:-

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

#define MAX 50

char stack[MAX];
int top = -1;

void push(char c) { stack[++top] = c; }
char pop() { return stack[top--]; }
char peek() { return stack[top]; }
int isEmpty() { return top == -1; }

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

// Infix to Postfix
void infixToPostfix(char* infix) {
    char postfix[MAX];
    int i, j = 0;

    for(i = 0; infix[i]; i++) {
        char c = infix[i];

        // Operand: add to output
        if(isalnum(c)) {
            postfix[j++] = c;
        }
        // '(': push
        else if(c == '(') {
            push(c);
        }
        // ')': pop until '('
        else if(c == ')') {
            while(!isEmpty() && peek() != '(')
                postfix[j++] = pop();
            pop(); // Remove '('
        }
        // Operator: pop higher/equal precedence, then push
        else {
            while(!isEmpty() && prec(peek()) >= prec(c))
                postfix[j++] = pop();
            push(c);
        }
    }

    // Pop remaining
```

```
    while(!isEmpty())
        postfix[j++] = pop();

    postfix[j] = '\0';
    printf("Postfix: %s\n", postfix);
}

int main() {
    char infix[MAX];
    printf("Infix: ");
    scanf("%s", infix);
    infixToPostfix(infix);
    return 0;
}
```

INFIXTOPREFIX:-

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

#define MAX 50

// Stack variables
char stack[MAX];
int top = -1;

// Stack functions
void push(char c) {
    stack[++top] = c;
}

char pop() {
    return stack[top--];
}

char peek() {
    return stack[top];
}

int isEmpty() {
    return top == -1;
}

// Get 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* str) {
    int len = strlen(str);
    for(int i = 0; i < len/2; i++) {
        char temp = str[i];
        str[i] = str[len-i-1];
        str[len-i-1] = temp;
    }
}

// Infix to Prefix conversion
void infixToPrefix(char* infix, char* prefix) {
    int i, j = 0;

    // Step 1: Reverse the infix
    reverse(infix);

    // Step 2: Swap ( and )
    for(i = 0; infix[i] != '\0'; i++) {
        if(infix[i] == '(')
            infix[i] = ')';
        else if(infix[i] == ')')
            infix[i] = '(';
    }

    // Step 3: Convert to postfix
    for(i = 0; infix[i] != '\0'; i++) {
        char c = infix[i];

        if(isalnum(c)) {
            prefix[j++] = c;
        }
        else if(c == '(') {
            push(c);
        }
        else if(c == ')') {
            while(!isEmpty() && peek() != '(') {
                prefix[j++] = pop();
            }
            pop();
        }
        else {
            // For ^: use >, others: use >=
            if(c == '^') {
                while(!isEmpty() && prec(peek()) > prec(c)) {
                    prefix[j++] = pop();
                }
            } else {
                while(!isEmpty() && prec(peek()) >= prec(c)) {
                    prefix[j++] = pop();
                }
            }
```

```
            push(c);
        }
    }

    while(!isEmpty()) {
        prefix[j++] = pop();
    }

    prefix[j] = '\0';

    // Step 4: Reverse the result
    reverse(prefix);
}

int main() {
    char infix[MAX], prefix[MAX];

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

    infixToPrefix(infix, prefix);

    printf("Prefix: %s\n", prefix);

    return 0;
}
```

POSTFIX EVALUATION :-

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

#define MAX 50

// Stack variables
int stack[MAX];
int top = -1;

// Stack functions
void push(int val) {
    stack[++top] = val;
}

int pop() {
    return stack[top--];
}

// Evaluate postfix expression
int evaluatePostfix(char* postfix) {
    int i;
```

```
    for(i = 0; postfix[i] != '\0'; i++) {
        char c = postfix[i];

        // If digit, push to stack
        if(isdigit(c)) {
            push(c - '0');
        }
        // If operator, pop 2 operands
        else {
            int op2 = pop();
            int op1 = pop();
            int result;

            if(c == '+') result = op1 + op2;
            else if(c == '-') result = op1 - op2;
            else if(c == '*') result = op1 * op2;
            else if(c == '/') result = op1 / op2;
            else if(c == '^') result = pow(op1, op2);

            push(result);
        }
    }

    return pop();
}

int main() {
    char postfix[MAX];

    printf("Enter postfix: ");
    scanf("%s", postfix);

    int result = evaluatePostfix(postfix);

    printf("Result: %d\n", result);

    return 0;
}
```

QUEUE (Array Implementation):-

```c
#include <stdio.h>
#define MAX 5

int arr[MAX];
int front = -1, rear = -1;

// Enqueue
void enqueue(int x) {
    if (rear == MAX - 1) {
        printf("Queue Full\n");
        return;
    }

    if (front == -1 && rear==-1){
        front =rear= 0;
    }else{
        rear++;
    }
    arr[rear] = x;
}

// Dequeue
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Empty\n");
        return;
    }

    printf("Removed %d\n", arr[front]);
    front++;

    // Reset when queue becomes empty
    if (front > rear)
        front = rear = -1;
}

// Display
```

```c
void display() {
    if (front == -1) {
        printf("Queue Empty\n");
        return;
    }

    for (int i = front; i <= rear; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int ch, x;

    while (1) {
        printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

QUEUE (Linked List Implementation):-

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

typedef struct Node
{
    int data;
    struct Node *link;
}node;

node *front=NULL,*rear=NULL;

void enque(int x){
    node *tmp=(node*)malloc(sizeof(node));
    tmp->data=x;
    tmp->link=NULL;

    if (rear == NULL)
    {
        front=rear=tmp;
    }else{
        rear->link=tmp;
        rear=tmp;
    }
}

void deque(){
    if (front == NULL)
    {
        printf("QUEUE UNDERFLOW");
        return;
    }
    node *tmp=(node*)malloc(sizeof(node));
    tmp=front;
    front=front->link;

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

    free(tmp);
}


void display() {
```

```
    node *ptr = front;
    if (ptr == NULL) {
        printf("Queue Empty\n");
        return;
    }

    while (ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

CIRCULAR QUEUE (Array Implementation):-

```
#include <stdio.h>
#define MAX 5

int cq[MAX];
int front = -1, rear = -1;

// Insert
void enqueue(int x) {
    if ((rear + 1) % MAX == front) {
        printf("Queue Full\n");
        return;
    }

    if (front == -1)
        front = rear = 0;
    else
        rear = (rear + 1) % MAX;
```

```c
    cq[rear] = x;
}

// Delete
void dequeue() {
    if (front == -1) {
        printf("Queue Empty\n");
        return;
    }

    printf("Removed %d\n", cq[front]);

    if (front == rear)
        front = rear = -1;
    else
        front = (front + 1) % MAX;
}

// Display
void display() {
    if (front == -1) {
        printf("Queue Empty\n");
        return;
    }

    printf("Queue elements: ");
    for (int i = front; ; i = (i + 1) % MAX) {
        printf("%d ", cq[i]);
        if (i == rear)
            break;
    }
    printf("\n");
}


int main() {
    int ch, x;
```

```
    while (1) {
        printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

CIRCULAR QUEUE (Linked List Implementation):-

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

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

Node *rear = NULL;

// Enqueue
void enqueue(int x) {
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->data = x;
```

```c
    if (rear == NULL) {
        rear = tmp;
        rear->next = rear;
    } else {
        tmp->next = rear->next;
        rear->next = tmp;
        rear = tmp;
    }
}

// Dequeue
void dequeue() {
    if (rear == NULL) {
        printf("Queue Empty\n");
        return;
    }

    Node *front = rear->next;

    printf("Removed %d\n", front->data);

    if (front == rear) {
        free(rear);
        rear = NULL;
    } else {
        rear->next = front->next;
        free(front);
    }
}

// Display
void display() {
    if (rear == NULL) {
        printf("Queue Empty\n");
        return;
    }

    Node *ptr = rear->next;
    do {
        printf("%d ", ptr->data);
        ptr = ptr->next;
    } while (ptr != rear->next);
    printf("\n");
}

int main() {
```

```
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

Doubly ended queue (Array implementation):-

```
#include <stdio.h>
#define MAX 5

int dq[MAX];
int front = -1, rear = -1;

// Insert at front
void insertFront(int x) {
    if (front == 0) {
        printf("Cannot insert at front\n");
        return;
    }
    if (front == -1)
        front = rear = 0;
    else
        front--;
    dq[front] = x;
}

// Insert at rear
void insertRear(int x) {
    if (rear == MAX - 1) {
        printf("Cannot insert at rear\n");
        return;
    }
    if (front == -1)
        front = rear = 0;
    else
        rear++;
    dq[rear] = x;
}

// Delete front
void deleteFront() {
```

```
    if (front == -1) {
        printf("Deque Empty\n");
        return;
    }
    printf("Deleted %d\n", dq[front]);
    if (front == rear)
        front = rear = -1;
    else
        front++;
}

// Delete rear
void deleteRear() {
    if (rear == -1) {
        printf("Deque Empty\n");
        return;
    }
    printf("Deleted %d\n", dq[rear]);
    if (front == rear)
        front = rear = -1;
    else
        rear--;
}

// Display
void display() {
    if (front == -1) {
        printf("Deque Empty\n");
        return;
    }
    for (int i = front; i <= rear; i++)
        printf("%d ", dq[i]);
    printf("\n");
}

int main() {
    insertRear(10);
    insertRear(20);
    insertFront(5);
    display();
    deleteFront();
    deleteRear();
    display();
    return 0;
}
```

OPERATIONS ON BST:-

INSERTION

DELETION

TRAVERSAL (PREORDER, INORDER, POSTORDER)

SEARCH

CODE:-

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

typedef struct Node
{
    int data;
    struct Node *l, *r;
} Node;
Node *newNode(int v)
{
    Node *n = malloc(sizeof(Node));
    n->data = v;
    n->l = n->r = NULL;
    return n;
}

Node *insert(Node *r, int v)
{
    if (!r)
        return newNode(v);
    if (v < r->data)
        r->l = insert(r->l, v);
    else if (v > r->data)
        r->r = insert(r->r, v);
    return r;
}

Node *search(Node *r, int k) {
    if(r==NULL || r->data==k) return r;
    return k < r->data ? search(r->l,k) : search(r->r,k);
```

```
}
void inorder(Node *r)
{
    if (r)
    {
        inorder(r->l);
        printf("%d ", r->data);
        inorder(r->r);
    }
}
void preorder(Node *r)
{
    if (r)
    {
        printf("%d ", r->data);
        preorder(r->l);
        preorder(r->r);
    }
}
void postorder(Node *r)
{
    if (r)
    {
        postorder(r->l);
        postorder(r->r);
        printf("%d ", r->data);
    }
}

Node *minNode(Node *r)
{
    while (r && r->l)
        r = r->l;
    return r;
}

Node *delNode(Node *r, int k)
{
```

```
    if (!r)
        return NULL;
    if (k < r->data)
        r->l = delNode(r->l, k);
    else if (k > r->data)
        r->r = delNode(r->r, k);
    else
    {
        if (!r->l && !r->r) // no child
        {
            free(r);
            return NULL;
        }
        if (!r->l)  // right child only
        {
            Node *t = r->r;
            free(r);
            return t;
        }
        if (!r->r) // left child only
        {
            Node *t = r->l;
            free(r);
            return t;
        }
        Node *s = minNode(r->r); // two children
        r->data = s->data;
        r->r = delNode(r->r, s->data);
    }
    return r;
}

int main()
{
    Node *root = NULL;
    int ch, x;
    while (1)
    {
```

```
        printf("\n1:Ins 2:In 3:Pre 4:Post 5:Sch 6:Del 7:Exit\n");
        if (scanf("%d", &ch) != 1)
            break;
        if (ch == 1)
        {
            scanf("%d", &x);
            root = insert(root, x);
        }
        else if (ch == 2)
        {
            inorder(root);
        }
        else if (ch == 3)
        {
            preorder(root);
        }
        else if (ch == 4)
        {
            postorder(root);
        }
        else if (ch == 5)
        {
            scanf("%d", &x);
            printf(search(root, x) ? "Found\n" : "Not Found\n");
        }
        else if (ch == 6)
        {
            scanf("%d", &x);
            root = delNode(root, x);
        }
        else
            break;
    }
    return 0;
}
```

1) Quick reminder: BST property

For any node X:

- All values in X->l are < X->data
- All values in X->r are > X->data

That property lets us search/delete in O(h) time by following left/right decisions.

---

2) The delete code (short version we used)

```
Node* delNode(Node* r,int k){
  if(!r) return NULL;
  if(k < r->data) r->l = delNode(r->l,k);
  else if(k > r->data) r->r = delNode(r->r,k);
  else{
    if(!r->l && !r->r){ free(r); return NULL; }      // no child
    if(!r->l){ Node* t=r->r; free(r); return t; }    // right child only
    if(!r->r){ Node* t=r->l; free(r); return t; }    // left child only
    Node* s = minNode(r->r); r->data = s->data;      // two children
    r->r = delNode(r->r, s->data);
  }
  return r;
}
```

And minNode:

```
Node* minNode(Node* r){ while(r && r->l) r=r->l; return r; }
```

We'll explain each line and the control flow with examples.

---

3) High-level idea of deletion cases

When we find the node N to delete:

1. **0 children (leaf)** — Just delete it and return NULL to the parent so parent's pointer becomes NULL.
2. **1 child** — Replace N with its single child: return the child pointer so parent now links to that child. Free N.
3. **2 children** — Choose a replacement value that preserves BST order. Typical choice: **inorder successor** (smallest node in N->r) or **inorder predecessor** (largest in N->l). We copy successor's value into N, then

delete the successor node from N->r (successor has at most one child). This keeps the tree structure valid.

---

4) Why use inorder successor (min of right subtree)?

- Inorder traversal of BST gives sorted sequence. For node N, its inorder successor S is the next larger value after N->data.
- S is the **leftmost** node in N->r. Because S is the smallest in the right subtree:
    - S->data > N->data and S has no left child.
    - Replace N->data with S->data still preserves BST ordering.
    - Deleting S is easier since S has at most one child (no left child).

Using predecessor (max in left subtree) is symmetric.

---

5) Detailed programmatic flow with an example

Example tree (inorder sorted: 20 30 40 50 60 70 80)

```
      50
    /    \
   30     70
  / \    / \
 20 40  60 80
```

Case A — delete a leaf (40)

Call delNode(root, 40):

- Compare at 50 → 40 < 50 → r->l = delNode(r->l,40) (call on subtree rooted at 30)
- At 30 → 40 > 30 → r->r = delNode(r->r,40) (call on subtree rooted at 40)
- At 40 → k == r->data → enters else
    - !r->l && !r->r true (leaf) → free(40) and return NULL
- Return back up: 30->r becomes NULL; then return 30 then 50.
  Result: 40 removed, parent pointers updated automatically by assignment r->r = delNode(...).

Important: **each recursive call returns the new subtree root**, which the parent assigns to its child pointer.

Case B — delete node with one child (delete 30 assuming 30 has only left child 20)

Modified tree:

```
  50
```

/ \

30 70

/ /\

20 60 80

Call delNode(root,30):

- At 50 → r->l = delNode(30,30)
- At 30 → found node; !r->l false, !r->r true (no right child)
    - Executes if(!r->r){ Node* t = r->l; free(r); return t; }
- Returns pointer to node 20 to parent (50), which assigns 50->l = 20.
  Result: 30 removed, 20 now child of 50.

Case C — delete node with two children (delete 50)

Original tree:

50

/ \

30 70

/ \ / \

20 40 60 80

Call delNode(root,50):

- At 50 → k == r->data → two children present.
- Node* s = minNode(r->r); → go to subtree 70 and walk left → finds 60.
- r->data = s->data; → copy 60 into root (50 becomes 60 logically).
  Tree now (values only):

60

/ \

30 70

/ \ \

20 40 80

- Then r->r = delNode(r->r, s->data); → delete node with value 60 in the original right subtree.
    - In r->r (which is subtree rooted at 70), call delNode(70,60).
    - At 70 → 60 < 70 → r->l = delNode(60,60).
    - At 60 node found; it's leaf (or has no left child), so freed and return NULL.

- Back at 70, 70->l becomes NULL; return 70; assign to root->r.
  Result: original 50 replaced by 60; original 60 node removed. BST property preserved.

---

6) Why recursion and return value matter

Note these lines:

if(k < r->data) r->l = delNode(r->l,k);

else if(k > r->data) r->r = delNode(r->r,k);

- Each call returns the **possibly updated subtree root** after deletion.
- The parent must reassign its l or r pointer to that returned pointer — because the root of that subtree might change (e.g., deleting the root of that subtree or replacing with child).
- This is the key to safe pointer updates without needing parent pointers.

---

7) Edge cases & safety

- **Deleting root**: works because top-level main holds root = delNode(root, k); and receives the new root.
- **Key not present**: recursion reaches r == NULL; function returns NULL up chain but parent pointers remain unchanged.
- **Memory safety**: we free() nodes when we remove them; we return appropriate child pointer to parent to avoid dangling pointers.
- **Duplicate keys**: typical BST assumes unique keys. If duplicates allowed, policy needed (e.g., store count or always go left/right).

---

8) Complexity

- Time = O(h) where h is tree height.
  - Balanced BST: h = O(log n) → O(log n)
  - Worst-case skewed BST: h = O(n) → O(n)
- Space due to recursion = O(h) stack frames.

Expression tree code:-

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

// Node of expression tree
typedef struct node {
    char data;
    struct node *left, *right;
} Node;

// Create node
Node* createNode(char x) {
    Node* n = (Node*)malloc(sizeof(Node));
    n->data = x;
    n->left = n->right = NULL;
    return n;
}

// Inorder traversal
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%c ", root->data);
        inorder(root->right);
    }
}

int main() {
    // Creating expression tree for (5 + 3) * 2

    Node* root = createNode('*');
```

```
    root->left = createNode('+');
    root->right = createNode('2');

    root->left->left = createNode('5');
    root->left->right = createNode('3');

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

    return 0;
}
```

GRAPH:-

1) Graph Representation — Adjacency Matrix (array)

Short, simple: build and print adjacency matrix for directed graph.

```
/* Adjacency Matrix (1..n) */
#include <stdio.h>
#define MAXV 100
int A[MAXV+1][MAXV+1];

int main(){
    int n, m;
    scanf("%d %d", &n, &m);            // n vertices, m edges
    for(int i=1;i<=n;i++) for(int j=1;j<=n;j++) A[i][j]=0;
    for(int e=0;e<m;e++){
        int u,v; scanf("%d %d",&u,&v); // directed edge u->v
        A[u][v]=1;
    }
    // print matrix
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++) printf("%d ", A[i][j]);
        printf("\n");
    }
    return 0;
}
```

- Complexity: building O(m), memory O(n^2).
- Use when graph is dense or professor expects matrix.

2) Graph Representation — Adjacency List (linked lists)

Build and print adjacency list. Easy to extend for traversals.

```
/* Adjacency List (directed) */
#include <stdio.h>
#include <stdlib.h>
#define MAXV 100

typedef struct Node { int v; struct Node *next; } Node;
Node* head[MAXV+1];

Node* newNode(int v){ Node* p=malloc(sizeof(Node)); p->v=v; p->next=NULL; return p; }

void addEdge(int u,int v){
    Node *p = newNode(v);
    p->next = head[u];
    head[u] = p;            // insert at front (exam-friendly)
}

int main(){
    int n,m; scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++) head[i]=NULL;
    for(int i=0;i<m;i++){ int u,v; scanf("%d %d",&u,&v); addEdge(u,v); }
```

```
    // print adjacency list
    for(int i=1;i<=n;i++){
        printf("%d: ", i);
        for(Node *p=head[i]; p; p=p->next) printf("%d -> ", p->v);
        printf("NULL\n");
    }
    return 0;
}
```

- Complexity: building O(n + m), memory O(n + m).
- Use for sparse graphs.

1 Depth-First Search (DFS)

Recursive DFS (Easiest version – best for exam)

```
#include <stdio.h>
#define MAX 20
int a[MAX][MAX], vis[MAX], n;

void dfs(int v){
    vis[v] = 1;
    printf("%d ", v);
    for(int i=1; i<=n; i++){
        if(a[v][i] == 1 && vis[i] == 0)
            dfs(i);
    }
}

int main(){
    int m, u, v, start;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &n, &m);

    for(int i=1;i<=m;i++){
        scanf("%d %d", &u, &v);
        a[u][v] = a[v][u] = 1;    // undirected
    }

    printf("Enter starting vertex: ");
    scanf("%d", &start);

    printf("DFS Traversal: ");
    dfs(start);
    return 0;
}
```

🔹 Logic (simple words):

1. Start from a node, mark it visited.
2. Recursively go to all **unvisited neighbors**.
3. When no neighbor remains, go back (recursion handles it).

🕓 **Complexity:** O(V + E)

2 Breadth-First Search (BFS)

BFS using Queue (standard and easiest)

```
#include <stdio.h>
#define MAX 20
int a[MAX][MAX], vis[MAX], q[MAX];
int front = 0, rear = -1, n;

void bfs(int start){
    vis[start] = 1;
    q[++rear] = start;

    while(front <= rear){
        int v = q[front++];
        printf("%d ", v);
        for(int i=1; i<=n; i++){
            if(a[v][i] == 1 && vis[i] == 0){
                vis[i] = 1;
                q[++rear] = i;
            }
        }
    }
}

int main(){
    int m,u,v,start;
    printf("Enter vertices and edges: ");
    scanf("%d %d",&n,&m);

    for(int i=1;i<=m;i++){
        scanf("%d %d",&u,&v);
        a[u][v]=a[v][u]=1;
    }

    printf("Enter start vertex: ");
    scanf("%d",&start);
    printf("BFS Traversal: ");
    bfs(start);
    return 0;
}
```

🔹 Logic:

1. Use a **queue** (FIFO).
2. Start node → enqueue → visit → enqueue all unvisited neighbors.
3. Continue until queue is empty.

🕒 **Complexity:** O(V + E)

SEARCHING TECHNIQUES

```c
#include <stdio.h>
#include <stdlib.h>
#define min(a, b) ((a < b) ? a : b)

void linearSearch(int arr[], int n, int key)
{
    int flag = 0, i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] == key)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
    {
        printf("ELEMENT %d FOUND at %d location\n", key, i + 1);
    }
    else
    {
        printf("ELEMENT %d NOT FOUND !\n", key);
    }
}

void BinarySearch(int arr[], int n, int key)
{
    int l = 0, r = n - 1, mid, flag = 0;

    while (l <= r)
    {
        mid = (l + r) / 2;
        if (arr[mid] == key)
        {
            flag = 1;
            break;
        }
        else if (arr[mid] < key)
        {
            r = mid - 1;
        }
        else if (arr[mid] > key)
        {
            l = mid + 1;
```

```
        }
    }
    if (flag == 1)
    {
        printf("ELEMENT %d FOUND at %d location\n", key, mid + 1);
    }
    else
    {
        printf("ELEMENT %d NOT FOUND !\n", key);
    }
}

void FibonacciSearch(int arr[], int n, int key)
{
    int f2 = 0, f1 = 1, f = f1 + f2, flag = 0, pos = -1, offset = -1;

    while (f < n)
    {
        f2 = f1;
        f1 = f;
        f = f1 + f2;
    }

    while (f > 1)
    {
        int i = min(offset + f2, n - 1);

        if (arr[i] < key)
        {
            f = f1;
            f1 = f2;
            f2 = f - f1;
            offset = i;
        }
        else if (arr[i] > key)
        {
            f = f2;
            f1 = f1 - f2;
            f2 = f - f1;
        }
        else
        {
            flag = 1;
            pos = i;
            break;
        }
```

```
    }

    if (!flag && f2 && arr[offset + 1] == key)
    {
        flag = 1;
        pos = offset + 1;
    }

    if (flag == 1)
    {
        printf("ELEMENT %d FOUND at %d location\n", key, pos + 1);
    }
    else
    {
        printf("ELEMENT %d NOT FOUND !\n", key);
    }
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("ARRAY: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t", arr[i]);
    }
    printf("\n");
    linearSearch(arr, n, 4);
    BinarySearch(arr, n, 10);
    FibonacciSearch(arr, n, 3);
    return 0;
}
```

SORTING TECHNIQUES

```c
#include <stdio.h>

/* Selection Sort - simple, in-place (unstable) */
void selectionSort(int a[], int n){
    int min,pos;
    for(int i=0;i<n;i++){
        min = a[i];
        pos=i;
        for (int j = i+1; j < n; j++){
            if (a[j]<min) {
                min = a[j];
                pos = j;
            }
        }
        a[pos]=a[i];
        a[i]=min;
    }
}

void insertionsort(int a[],int n){
    int key,j;
    for (int  i = 1; i < n; i++){
        key=a[i];
        j=i-1;
        while (j>=0 && a[j]>key){
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=key;
    }
}
void merge(int a[],int l,int mid,int h){
    int n1=mid-l+1;
    int n2=h-mid;
    int L[n1],R[n2];

    for(int i=0;i<n1;i++) L[i]=a[l+i];
    for(int i=0;i<n2;i++) R[i]=a[mid+1+i];

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

    while (i<n1 && j<n2){
        if (L[i]<=R[j]){
            a[k]=L[i];
```

```
            i++;
        }else{
            a[k]=R[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k]=L[i];
        i++;
        k++;
    }
    while (j<n2)
    {
        a[k]=R[j];
        j++;
        k++;
    }
}
void mergesort(int a[],int l,int h){
    if (l<h)
    {
        int mid=(l+h)/2;
        mergesort(a,l,mid);
        mergesort(a,mid+1,h);
        merge(a,l,mid,h);
    }
}

int partition(int a[],int l,int h){
    int pivot=a[l];
    int i=l,j=h;

    while (i<j)
    {
        while (i <= h && a[i]<=pivot)
        {
            i++;
        }
        while (a[j]>pivot)
        {
            j--;
        }
        if (i<j)
        {
```

```
            int temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    a[l]=a[j];
    a[j]=pivot;
    return j;
}
void quickSort(int a[], int l, int h) {
    if (l<h)
    {
        int loc=partition(a,l,h);
        quickSort(a,l,loc-1);
        quickSort(a,loc+1,h);
    }
}


int getMax(int a[], int n) {
    int max = a[0];
    for(int i = 1; i < n; i++)
        if(a[i] > max) max = a[i];
    return max;
}

void radixSort(int a[], int n) {
    int max = getMax(a, n), exp = 1, out[50];

    while(max / exp > 0) {
        int count[10] = {0};

        for(int i = 0; i < n; i++)
            count[(a[i] / exp) % 10]++;

        for(int i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for(int i = n - 1; i >= 0; i--)
            out[--count[(a[i] / exp) % 10]] = a[i];

        for(int i = 0; i < n; i++)
            a[i] = out[i];

        exp *= 10;
    }
```

```
}

int main(){
    int a[8] = {10, 70, 20, 80, 30, 40, 50, 60};
    int n = sizeof(a)/sizeof(a[0]);
    printf("BEFORE: ");
    for(int i=0;i<n;i++) printf("%d ", a[i]);
    // selectionSort(a,n);
    // insertionsort(a,n);
    // mergesort(a,0,n-1);
    // quickSort(a, 0, n-1);
    // radixSort(a,n);
    printf("\nAFTER: ");
    for(int i=0;i<n;i++) printf("%d ", a[i]);
    return 0;
}
```

HEAP SORT:-

```
#include <stdio.h>

void heapify(int a[], int n, int i) {
   int small = i;
   int l = 2*i + 1;
   int r = 2*i + 2;

   if (l < n && a[l] < a[small]) small = l;
   if (r < n && a[r] < a[small]) small = r;

   if (small != i) {
      int t = a[i]; a[i] = a[small]; a[small] = t;
      heapify(a, n, small);
   }
```

```
}

int main() {
   int a[] = {4, 1, 3, 2};
   int n = 4;

   // Build MIN heap
   for (int i = n/2 - 1; i >= 0; i--)
      heapify(a, n, i);

   printf("Sorted order: ");
   while (n > 0) {
      printf("%d ", a[0]);     // root = smallest
      a[0] = a[n-1];          // move last to root
      n--;
      heapify(a, n, 0);
   }

   return 0;
}
```

Min Heap → root = smallest → delete → sorted