# Ex. No. 1a ARRAY IMPLEMENTATION OF STACK

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

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

void push(int x) {
    if (top < MAX - 1) {
        stack[++top] = x;
        printf("\n%d pushed to stack\n", x);
    } else {
        printf("\nStack Overflow\n");
    }
}

int pop() {
    if (top >= 0) {
        int popped_value = stack[top--];
        printf("\n%d popped from stack\n", popped_value);
        return popped_value;
    } else {
        printf("\nStack Underflow\n");
        return -1;  // Return an error value
    }
}

void view() {
    if (top < 0) {
        printf("\nStack is empty\n");
    } else {
        printf("\nStack elements (Top to Bottom):\n");
        for (int i = top; i >= 0; i--) {
            printf("%4d\n", stack[i]);
        }
    }
}

int main() {
    int ch = 0, val;

    while (ch != 4) {
        printf("\nSTACK OPERATIONS\n");
        printf("1. PUSH\n");
        printf("2. POP\n");
        printf("3. VIEW\n");
        printf("4. QUIT\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
```

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

            case 2:
                pop();
                break;

            case 3:
                view();
                break;

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

            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```

## Ex. No. 1b ARRAY IMPLEMENTATION OF QUEUE

```
#include <stdio.h>
#define MAX 5  // Define the maximum size of the queue

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

// Function to insert an element into the queue
void enqueue(int value) {
    if (rear == MAX - 1) {
        // Check if the queue is full
        printf("\nQueue is full! Insertion not possible.\n");
    } else {
        if (front == -1) {
            // If the queue is initially empty
            front = 0;
        }
        rear++;
```

```
        queue[rear] = value;
        printf("\nInserted %d into the queue.\n", value);
    }
}

// Function to remove an element from the queue
int dequeue() {
    if (front == -1 || front > rear) {
        // Check if the queue is empty
        printf("\nQueue is empty! Deletion not possible.\n");
        return -1;
    } else {
        int removed_value = queue[front];
        front++;
        if (front > rear) {
            // If the queue becomes empty after dequeuing
            front = rear = -1;
        }
        printf("\nRemoved %d from the queue.\n", removed_value);
        return removed_value;
    }
}

// Function to display the elements in the queue
void display() {
    if (front == -1) {
        printf("\nQueue is empty!\n");
    } else {
        printf("\nQueue elements:\nFront --> ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("<-- Rear\n");
    }
}

// Main function to demonstrate queue operations
int main() {
    int choice, value;

    while (1) {
        printf("\nQUEUE OPERATIONS:\n");
        printf("1. Enqueue (Insert)\n");
        printf("2. Dequeue (Remove)\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
            printf("Enter value to insert: ");
            scanf("%d", &value);
            enqueue(value);
            break;

        case 2:
            dequeue();
            break;

        case 3:
            display();
            break;

        case 4:
            printf("Exiting...\n");
            return 0;

        default:
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

## Ex. 2 Array implementation of List

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

void create();
void insert();
void deletion();
void search();
void display();

int b[MAX], n = 0, p, e, pos, i;

int main() {
    int ch;
    char g = 'y';

    do {
        printf("\nMain Menu");
        printf("\n1. Create");
        printf("\n2. Delete");
        printf("\n3. Search");
        printf("\n4. Insert");
```

```
        printf("\n5. Display");
        printf("\n6. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);

        switch(ch) {
            case 1:
                create();
                break;
            case 2:
                deletion();
                break;
            case 3:
                search();
                break;
            case 4:
                insert();
                break;
            case 5:
                display();
                break;
            case 6:
                exit(0);
                break;
            default:
                printf("\nInvalid choice. Please enter a correct choice.");
        }

        printf("\nDo you want to continue (y/n): ");
        scanf(" %c", &g); // Note the space before %c to consume any leftover newline character
    } while (g == 'y' || g == 'Y');

    return 0;
}

void create() {
    printf("\nEnter the number of elements (max %d): ", MAX);
    scanf("%d", &n);
    if (n > MAX) {
        printf("\nExceeded maximum limit of %d elements. Setting n to %d.\n", MAX, MAX);
        n = MAX;
    }
    for (i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &b[i]);
    }
}

void deletion() {
```

```
    printf("\nEnter the position you want to delete (1 to %d): ", n);
    scanf("%d", &pos);
    if (pos < 1 || pos > n) {
        printf("\nInvalid position.");
    } else {
        for (i = pos - 1; i < n - 1; i++) {
            b[i] = b[i + 1];
        }
        n--;
        printf("\nThe elements after deletion:");
        display();
    }
}

void search() {
    printf("\nEnter the element to be searched: ");
    scanf("%d", &e);
    for (i = 0; i < n; i++) {
        if (b[i] == e) {
            printf("Value %d is at position %d.\n", e, i + 1);
            return;
        }
    }
    printf("Value %d is not in the list.\n", e);
}

void insert() {
    if (n >= MAX) {
        printf("\nCannot insert more elements, list is full.");
        return;
    }
    printf("\nEnter the position where you need to insert (1 to %d): ", n + 1);
    scanf("%d", &pos);
    if (pos < 1 || pos > n + 1) {
        printf("\nInvalid position.");
    } else {
        for (i = n; i >= pos - 1; i--) {
            b[i + 1] = b[i];
        }
        printf("\nEnter the element to insert: ");
        scanf("%d", &p);
        b[pos - 1] = p;
        n++;
        printf("\nThe list after insertion:");
        display();
    }
}

void display() {
```

```
    if (n == 0) {
        printf("\nThe list is empty.");
    } else {
        printf("\nThe elements of the list are:");
        for (i = 0; i < n; i++) {
            printf("\n%d", b[i]);
        }
    }
}
```

## EX. 3a LINKED LIST IMPLEMENTATION OF LIST [SINGLY LINKED LIST]

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

// Define the structure of a node in the singly linked list
struct node {
    int label;
    struct node *next;
};

// Function declarations
void addNode(struct node *head, int k);
void deleteNode(struct node *head, int k);
void viewList(struct node *head);

int main() {
    int ch, k;
    struct node *head = (struct node*) malloc(sizeof(struct node));
    head->label = -1;
    head->next = NULL;

    while (1) {
        printf("\n\nSINGLY LINKED LIST OPERATIONS\n");
        printf("1 -> Add\n");
        printf("2 -> Delete\n");
        printf("3 -> View\n");
        printf("4 -> Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("\nEnter label after which to add: ");
                scanf("%d", &k);
                addNode(head, k);
                break;
            case 2:
                printf("\nEnter label of node to be deleted: ");
```

```
                scanf("%d", &k);
                deleteNode(head, k);
                break;
            case 3:
                viewList(head);
                break;
            case 4:
                printf("\nExiting...\n");
                exit(0);
                break;
            default:
                printf("\nInvalid choice. Please try again.\n");
        }
    }
    return 0;
}

// Function to add a node after a node with a given label
void addNode(struct node *head, int k) {
    struct node *h = head;

    // Traverse the list to find the node with label `k`
    while (h != NULL && h->label != k) {
        h = h->next;
    }

    if (h == NULL) {
        printf("Node with label %d not found\n", k);
    } else {
        struct node *temp = (struct node*) malloc(sizeof(struct node));
        printf("Enter label for new node: ");
        scanf("%d", &temp->label);
        temp->next = h->next;
        h->next = temp;
        printf("Node added successfully\n");
    }
}

// Function to delete a node with a given label
void deleteNode(struct node *head, int k) {
    struct node *prev = head;
    struct node *curr = head->next;
    int found = 0;

    // Traverse to find the node with label k
    while (curr != NULL) {
        if (curr->label == k) {
            found = 1;
            break;
```

```
        }
        prev = curr;
        curr = curr->next;
    }

    if (found) {
        prev->next = curr->next;
        free(curr);
        printf("Node deleted successfully\n");
    } else {
        printf("Sorry, node with label %d not found\n", k);
    }
}

// Function to display the list
void viewList(struct node *head) {
    struct node *h = head->next; // Skip the head node as it's a dummy
    if (h == NULL) {
        printf("\nThe list is empty\n");
        return;
    }
    printf("\nHEAD -> ");
    while (h != NULL) {
        printf("%d -> ", h->label);
        h = h->next;
    }
    printf("NULL\n");
}
```

## Ex. No. 3b LINKED LIST IMPLEMENTATION OF STACK

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

// Define the structure of a node in the singly linked list
struct node {
    int label;
    struct node *next;
};

// Function declarations
void push(struct node *head, int label);
void pop(struct node *head);
void viewStack(struct node *head);

int main() {
    int ch, label;
    struct node *head = (struct node*) malloc(sizeof(struct node));
    head->next = NULL;
```

```
    while(1) {
        printf("\nStack using Linked List\n");
        printf("1 -> Push\n");
        printf("2 -> Pop\n");
        printf("3 -> View\n");
        printf("4 -> Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch(ch) {
            case 1:
                printf("Enter label for new node: ");
                scanf("%d", &label);
                push(head, label);
                break;
            case 2:
                pop(head);
                break;
            case 3:
                viewStack(head);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

// Function to push a new node onto the stack
void push(struct node *head, int label) {
    struct node *temp = (struct node*) malloc(sizeof(struct node));
    temp->label = label;
    temp->next = head->next;
    head->next = temp;
    printf("Node with label %d pushed onto the stack.\n", label);
}

// Function to pop the top node from the stack
void pop(struct node *head) {
    if (head->next == NULL) {
        printf("Stack is empty. Cannot pop.\n");
        return;
    }
    struct node *temp = head->next;
    head->next = temp->next;
```

```
    printf("Node with label %d popped from the stack.\n", temp->label);
    free(temp);
}

// Function to view the stack
void viewStack(struct node *head) {
    if (head->next == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    struct node *h = head->next;
    printf("\nStack (top to bottom):\n");
    while (h != NULL) {
        printf("%d -> ", h->label);
        h = h->next;
    }
    printf("NULL\n");
}
```

OUTPUT

```
Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: 1
Enter label for new node: 25
Node with label 25 pushed onto the stack.

Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: 3

Stack (top to bottom):
25 -> NULL

Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: 1
Enter label for new node: 25 57
Node with label 25 pushed onto the stack.
```

```
Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: Invalid choice. Please try again.

Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: 3

Stack (top to bottom):
25 -> 25 -> NULL

Stack using Linked List
1 -> Push
2 -> Pop
3 -> View
4 -> Exit
Enter your choice: 4
Exiting...
```

**Ex. No. 3c LINKED LIST IMPLEMENTATION OF QUEUE**

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

// Define the structure of a node in the singly linked list
struct node {
    int label;
    struct node *next;
};

// Function declarations
void insertNode(struct node *head, int label);
void deleteNode(struct node *head);
void viewQueue(struct node *head);

int main() {
    int ch, label;
    struct node *head = (struct node*) malloc(sizeof(struct node));
    head->next = NULL;

    while(1) {
        printf("\nQueue using Linked List\n");
        printf("1 -> Insert\n");
        printf("2 -> Delete\n");
```

```
        printf("3 -> View\n");
        printf("4 -> Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch(ch) {
            case 1:
                printf("Enter label for new node: ");
                scanf("%d", &label);
                insertNode(head, label);
                break;
            case 2:
                deleteNode(head);
                break;
            case 3:
                viewQueue(head);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

// Function to insert a node at the end of the queue
void insertNode(struct node *head, int label) {
    struct node *temp = (struct node*) malloc(sizeof(struct node));
    temp->label = label;
    temp->next = NULL;

    struct node *h = head;
    while (h->next != NULL) {
        h = h->next;
    }
    h->next = temp;
    printf("Node with label %d inserted into the queue.\n", label);
}

// Function to delete the front node from the queue
void deleteNode(struct node *head) {
    if (head->next == NULL) {
        printf("Queue is empty. Cannot delete.\n");
        return;
    }

    struct node *temp = head->next;
```

```
    head->next = temp->next;
    printf("Node with label %d deleted from the queue.\n", temp->label);
    free(temp);
}

// Function to view the queue
void viewQueue(struct node *head) {
    if (head->next == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    struct node *h = head->next;
    printf("\nQueue (front to rear):\n");
    while (h != NULL) {
        printf("%d -> ", h->label);
        h = h->next;
    }
    printf("NULL\n");
}
```

OUTPUT

```
Queue using Linked List
1 -> Insert
2 -> Delete
3 -> View
4 -> Exit
Enter your choice: 1
Enter label for new node: 12
Node with label 12 inserted into the queue.

Queue using Linked List
1 -> Insert
2 -> Delete
3 -> View
4 -> Exit
Enter your choice: 1
Enter label for new node: 13
Node with label 13 inserted into the queue.

Queue using Linked List
1 -> Insert
2 -> Delete
3 -> View
4 -> Exit
Enter your choice: 3

Queue (front to rear):
```

12 -> 13 -> NULL

Queue using Linked List
1 -> Insert
2 -> Delete
3 -> View
4 -> Exit
Enter your choice: 4
Exiting...

## Ex. No. 4a APPLICATIONS OF LIST

### POLYNOMIAL ADDITION AND SUBTRACTION

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

struct node {
    int num;
    int coeff;
    struct node *next;
};

struct node *start1 = NULL;
struct node *start2 = NULL;
struct node *start3 = NULL;
struct node *start4 = NULL;

struct node *create_poly(struct node *);
void display_poly(struct node *);
struct node *add_poly(struct node *, struct node *, struct node *);
struct node *sub_poly(struct node *, struct node *, struct node *);
struct node *add_node(struct node *, int, int);

int main() {
    int option;

    do {
        printf("\n******* MAIN MENU *******\n");
        printf("1. Enter the first polynomial\n");
        printf("2. Display the first polynomial\n");
        printf("3. Enter the second polynomial\n");
        printf("4. Display the second polynomial\n");
        printf("5. Add the polynomials\n");
        printf("6. Display the addition result\n");
        printf("7. Subtract the polynomials\n");
        printf("8. Display the subtraction result\n");
        printf("9. EXIT\n");
        printf("Enter your option: ");
        scanf("%d", &option);
```

```
        switch(option) {
            case 1:
                start1 = create_poly(start1);
                break;
            case 2:
                display_poly(start1);
                break;
            case 3:
                start2 = create_poly(start2);
                break;
            case 4:
                display_poly(start2);
                break;
            case 5:
                start3 = add_poly(start1, start2, start3);
                break;
            case 6:
                display_poly(start3);
                break;
            case 7:
                start4 = sub_poly(start1, start2, start4);
                break;
            case 8:
                display_poly(start4);
                break;
        }
    } while(option != 9);

    return 0;
}

struct node *create_poly(struct node *start) {
    struct node *new_node, *ptr;
    int n, c;

    printf("\nEnter the number (-1 to end): ");
    scanf("%d", &n);
    while(n != -1) {
        printf("Enter its coefficient: ");
        scanf("%d", &c);

        new_node = (struct node *)malloc(sizeof(struct node));
        new_node->num = n;
        new_node->coeff = c;
        new_node->next = NULL;

        if(start == NULL) {
            start = new_node;
        } else {
            ptr = start;
```

```
            while(ptr->next != NULL) {
                ptr = ptr->next;
            }
            ptr->next = new_node;
        }

        printf("\nEnter the number (-1 to end): ");
        scanf("%d", &n);
    }

    return start;
}

void display_poly(struct node *start) {
    struct node *ptr = start;
    while(ptr != NULL) {
        printf("%d x^%d", ptr->num, ptr->coeff);
        if(ptr->next != NULL) {
            printf(" + ");
        }
        ptr = ptr->next;
    }
    printf("\n");
}

struct node *add_poly(struct node *start1, struct node *start2, struct node *start3) {
    struct node *ptr1 = start1, *ptr2 = start2;

    while(ptr1 != NULL && ptr2 != NULL) {
        if(ptr1->coeff == ptr2->coeff) {
            start3 = add_node(start3, ptr1->num + ptr2->num, ptr1->coeff);
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        } else if(ptr1->coeff > ptr2->coeff) {
            start3 = add_node(start3, ptr1->num, ptr1->coeff);
            ptr1 = ptr1->next;
        } else {
            start3 = add_node(start3, ptr2->num, ptr2->coeff);
            ptr2 = ptr2->next;
        }
    }

    while(ptr1 != NULL) {
        start3 = add_node(start3, ptr1->num, ptr1->coeff);
        ptr1 = ptr1->next;
    }

    while(ptr2 != NULL) {
        start3 = add_node(start3, ptr2->num, ptr2->coeff);
        ptr2 = ptr2->next;
    }
```

```
    return start3;
}

struct node *sub_poly(struct node *start1, struct node *start2, struct node *start4) {
    struct node *ptr1 = start1, *ptr2 = start2;

    while(ptr1 != NULL && ptr2 != NULL) {
        if(ptr1->coeff == ptr2->coeff) {
            start4 = add_node(start4, ptr1->num - ptr2->num, ptr1->coeff);
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        } else if(ptr1->coeff > ptr2->coeff) {
            start4 = add_node(start4, ptr1->num, ptr1->coeff);
            ptr1 = ptr1->next;
        } else {
            start4 = add_node(start4, ptr2->num, ptr2->coeff);
            ptr2 = ptr2->next;
        }
    }

    while(ptr1 != NULL) {
        start4 = add_node(start4, ptr1->num, ptr1->coeff);
        ptr1 = ptr1->next;
    }

    while(ptr2 != NULL) {
        start4 = add_node(start4, ptr2->num, ptr2->coeff);
        ptr2 = ptr2->next;
    }

    return start4;
}

struct node *add_node(struct node *start, int n, int c) {
    struct node *new_node, *ptr;

    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->num = n;
    new_node->coeff = c;
    new_node->next = NULL;

    if(start == NULL) {
        start = new_node;
    } else {
        ptr = start;
        while(ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = new_node;
    }
```

```
    return start;
}
```

**Ex. No. 5 A INFIX TO POSTFIX**

```
#include <stdio.h>
#include <string.h>
#define MAX 20

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

void push(char item);
char pop();
int prcd(char symbol);
int isoperator(char symbol);
void convertip(char infix[], char postfix[]);

int main() {
    char infix[20], postfix[20];

    printf("Enter the valid infix string: ");
    fgets(infix, 20, stdin);
    // Remove the newline character if present
    infix[strcspn(infix, "\n")] = '\0';

    convertip(infix, postfix);
    printf("The corresponding postfix string is: ");
    puts(postfix);

    return 0;
}

void push(char item) {
    if (top < MAX - 1) {
        stack[++top] = item;
    } else {
        printf("Stack overflow\n");
    }
}

char pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
        printf("Stack underflow\n");
        return '\0';
    }
}
```

```
int prcd(char symbol) {
    switch (symbol) {
        case '+':
        case '-':
            return 2;
        case '*':
        case '/':
            return 4;
        case '^':
        case '$':
            return 6;
        case '(':
        case ')':
        case '#':
            return 1;
        default:
            return 0;
    }
}

int isoperator(char symbol) {
    switch (symbol) {
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '$':
        case '(':
        case ')':
            return 1;
        default:
            return 0;
    }
}

void convertip(char infix[], char postfix[]) {
    int i, symbol, j = 0;
    stack[++top] = '#';

    for (i = 0; i < strlen(infix); i++) {
        symbol = infix[i];
        if (isoperator(symbol) == 0) {
            postfix[j++] = symbol;
        } else {
            if (symbol == '(') {
                push(symbol);
            } else if (symbol == ')') {
                while (stack[top] != '(') {
                    postfix[j++] = pop();
                }
```

```
            pop();  // Pop out '('.
        } else {
            while (prcd(symbol) <= prcd(stack[top])) {
                postfix[j++] = pop();
            }
            push(symbol);
        }
    }
}

while (stack[top] != '#') {
    postfix[j++] = pop();
}

postfix[j] = '\0';
}
```

## Ex. No. 5 B Expression Evaluation

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

struct stack {
    float a[50];
    int top;
} s;

int main() {
    char pf[50];
    float d1, d2;
    int i;

    s.top = -1;

    printf("\n\nEnter the postfix expression: ");
    fgets(pf, 50, stdin);
    // Remove the newline character if present
    pf[strcspn(pf, "\n")] = '\0';

    for (i = 0; pf[i] != '\0'; i++) {
        switch (pf[i]) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                s.a[++s.top] = pf[i] - '0';
                break;
            case '+':
                d1 = s.a[s.top--];
                d2 = s.a[s.top--];
                s.a[++s.top] = d1 + d2;
                break;
```

```
        case '-':
            d2 = s.a[s.top--];
            d1 = s.a[s.top--];
            s.a[++s.top] = d1 - d2;
            break;
        case '*':
            d2 = s.a[s.top--];
            d1 = s.a[s.top--];
            s.a[++s.top] = d1 * d2;
            break;
        case '/':
            d2 = s.a[s.top--];
            d1 = s.a[s.top--];
            if (d2 != 0)
                s.a[++s.top] = d1 / d2;
            else
                printf("\nError: Division by zero");
            break;
        default:
            printf("\nError: Invalid character '%c' in expression\n", pf[i]);
            return 1;
      }
    }

    printf("\nExpression value is: %.2f\n", s.a[s.top]);

    return 0;
}
```

OUTPUT
Enter the postfix expression: 6523+8*+3+*

Expression value is: 288.00

**Ex No. 6 IMPLEMENTATION BINARY SEARCH TREE**

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

struct searchtree {
    int element;
    struct searchtree *left, *right;
} *root;

typedef struct searchtree *node;
typedef int ElementType;

node insert(ElementType, node);
node delete(ElementType, node);
void makeempty();
node findmin(node);
```

```
node findmax(node);
node find(ElementType, node);
void display(node, int);

int main() {
    int ch;
    ElementType a;
    node temp;

    makeempty();

    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Find\n4. Find min\n5. Find max\n6. Display\n7. Exit\nEnter Your
Choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter an element: ");
                scanf("%d", &a);
                root = insert(a, root);
                break;
            case 2:
                printf("\nEnter the element to delete: ");
                scanf("%d", &a);
                root = delete(a, root);
                break;
            case 3:
                printf("\nEnter the element to search: ");
                scanf("%d", &a);
                temp = find(a, root);
                if (temp != NULL)
                    printf("Element found\n");
                else
                    printf("Element not found\n");
                break;
            case 4:
                temp = findmin(root);
                if (temp == NULL)
                    printf("\nEmpty tree\n");
                else
                    printf("\nMinimum element: %d\n", temp->element);
                break;
            case 5:
                temp = findmax(root);
                if (temp == NULL)
                    printf("\nEmpty tree\n");
                else
                    printf("\nMaximum element: %d\n", temp->element);
                break;
            case 6:
                if (root == NULL)
```

```
                printf("\nEmpty tree\n");
            else
                display(root, 1);
            break;
        case 7:
            exit(0);
        default:
            printf("Invalid Choice\n");
        }
    }
    return 0;
}

node insert(ElementType x, node t) {
    if (t == NULL) {
        t = (node)malloc(sizeof(struct searchtree));
        if (t == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }
        t->element = x;
        t->left = t->right = NULL;
    } else {
        if (x < t->element)
            t->left = insert(x, t->left);
        else if (x > t->element)
            t->right = insert(x, t->right);
    }
    return t;
}

node delete(ElementType x, node t) {
    node temp;
    if (t == NULL) {
        printf("\nElement not found\n");
    } else if (x < t->element) {
        t->left = delete(x, t->left);
    } else if (x > t->element) {
        t->right = delete(x, t->right);
    } else {
        if (t->left != NULL && t->right != NULL) {
            temp = findmin(t->right);
            t->element = temp->element;
            t->right = delete(t->element, t->right);
        } else {
            temp = t;
            if (t->left == NULL)
                t = t->right;
            else
                t = t->left;
            free(temp);
```

```
        }
    }
    return t;
}

void makeempty() {
    root = NULL;
}

node findmin(node t) {
    if (t == NULL)
        return NULL;
    else if (t->left == NULL)
        return t;
    else
        return findmin(t->left);
}

node findmax(node t) {
    if (t == NULL)
        return NULL;
    else if (t->right == NULL)
        return t;
    else
        return findmax(t->right);
}

node find(ElementType x, node t) {
    if (t == NULL)
        return NULL;
    if (x < t->element)
        return find(x, t->left);
    else if (x > t->element)
        return find(x, t->right);
    else
        return t;
}

void display(node t, int level) {
    int i;
    if (t != NULL) {
        display(t->right, level + 1);
        printf("\n");
        for (i = 0; i < level; i++)
            printf("    ");
        printf("%d", t->element);
        display(t->left, level + 1);
    }
}
```

OUPUT

```
/tmp/atDrF0LQnH.o

1. Insert
2. Delete
3. Find
4. Find min
5. Find max
6. Display
7. Exit
Enter Your Choice: 1
Enter an element: 10

1. Insert
2. Delete
3. Find
4. Find min
5. Find max
6. Display
7. Exit
Enter Your Choice: 1
Enter an element: 58

1. Insert
2. Delete
3. Find
4. Find min
5. Find max
6. Display
7. Exit
Enter Your Choice: 1
Enter an element: 65

1. Insert
2. Delete
3. Find
4. Find min
5. Find max
6. Display
7. Exit
Enter Your Choice: 6

            65
        58
    10
1. Insert
2. Delete
3. Find
4. Find min
5. Find max
6. Display
```

7. Exit
Enter Your Choice:

**Ex. No.7 IMPLEMENTATION OF AVL TREES**

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

// An AVL tree node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

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

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

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    if (node == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
struct Node *rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;

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

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

    // Return new root
    return x;
}

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

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

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

    // Return new root
    return y;
}

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

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

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

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

    // 3. Get the balance factor of this ancestor
```

```
    // node to check whether this node became
    // unbalanced
    int balance = getBalance(node);

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

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

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

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

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

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

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main() {
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
```

```
    root = insert(root, 50);
    root = insert(root, 25);
    printf("Preorder traversal of the constructed AVL tree is \n");
    preOrder(root);

    return 0;
}
```

**Ex. No.8 IMPLEMENTATION OF HEAP USING PRIORITY QUEUES**

```
#include <stdio.h>

#define TREE_ARRAY_SIZE 20
int heap_size = 0;
const int INF = 100000;

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

// Function to get the right child of a node
int get_right_child(int index) {
    if ((((2 * index) + 1) < TREE_ARRAY_SIZE && index >= 0)
        return (2 * index) + 1;
    return -1;
}

// Function to get the left child of a node
int get_left_child(int index) {
    if ((2 * index) < TREE_ARRAY_SIZE && index >= 0)
        return 2 * index;
    return -1;
}

// Function to get the parent of a node
int get_parent(int index) {
    if (index > 0 && index < TREE_ARRAY_SIZE)
        return (index - 1) / 2;
    return -1;
}

void max_heapify(int A[], int index) {
    int left_child_index = get_left_child(index);
    int right_child_index = get_right_child(index);
    int largest = index;

    if (left_child_index < heap_size && A[left_child_index] > A[largest]) {
        largest = left_child_index;
    }
```

```
    if (right_child_index < heap_size && A[right_child_index] > A[largest]) {
        largest = right_child_index;
    }

    if (largest != index) {
        swap(&A[index], &A[largest]);
        max_heapify(A, largest);
    }
}

void build_max_heap(int A[]) {
    for (int i = heap_size / 2 - 1; i >= 0; i--) {
        max_heapify(A, i);
    }
}

int maximum(int A[]) {
    if (heap_size > 0)
        return A[0];
    printf("Heap is empty\n");
    return -1;
}

int extract_max(int A[]) {
    if (heap_size < 1) {
        printf("Heap underflow\n");
        return -1;
    }

    int maxm = A[0];
    A[0] = A[heap_size - 1];
    heap_size--;
    max_heapify(A, 0);
    return maxm;
}

void increase_key(int A[], int index, int key) {
    if (key < A[index]) {
        printf("New key is smaller than current key\n");
        return;
    }

    A[index] = key;
    while (index > 0 && A[get_parent(index)] < A[index]) {
        swap(&A[index], &A[get_parent(index)]);
        index = get_parent(index);
    }
}

void decrease_key(int A[], int index, int key) {
```

```c
    if (key > A[index]) {
        printf("New key is larger than current key\n");
        return;
    }

    A[index] = key;
    max_heapify(A, index);
}

void insert(int A[], int key) {
    if (heap_size >= TREE_ARRAY_SIZE) {
        printf("Heap overflow\n");
        return;
    }

    heap_size++;
    A[heap_size - 1] = -INF; // Initialize with a very small value
    increase_key(A, heap_size - 1, key);
}

int main() {
    int A[TREE_ARRAY_SIZE] = {0}; // Initialize the array with zeros

    // Insert elements
    insert(A, 10);
    insert(A, 20);
    insert(A, 30);
    insert(A, 40);
    insert(A, 50);

    // Build max heap
    build_max_heap(A);

    printf("Maximum value: %d\n", maximum(A));
    printf("Extracted maximum: %d\n", extract_max(A));
    printf("Maximum value after extraction: %d\n", maximum(A));

    return 0;
}
```

OUTPUT

```
Maximum value: 50
Extracted maximum: 50
Maximum value after extraction: 40
```

**Ex. No. 9 DIJKSTRA'S ALGORITHM**

```c
#include <stdio.h>
#include <limits.h> // for INT_MAX
```

```
#define MAX 10
#define INFINITY INT_MAX

void dijkstra(int G[MAX][MAX], int n, int startnode);

int main() {
    int G[MAX][MAX];
    int i, j, n, u;

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

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

    printf("\nEnter the starting node: ");
    scanf("%d", &u);

    dijkstra(G, n, u);

    return 0;
}

void dijkstra(int G[MAX][MAX], int n, int startnode) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX];
    int count, mindistance, nextnode, i, j;

    // Initialize the cost matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (G[i][j] == 0) {
                cost[i][j] = (i == j) ? 0 : INFINITY; // No path if G[i][j] == 0 except diagonal
            } else {
                cost[i][j] = G[i][j];
            }
        }
    }

    // Initialize pred[], distance[], and visited[]
    for (i = 0; i < n; i++) {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }

    distance[startnode] = 0;
```

```
    visited[startnode] = 1;
    count = 1;

    while (count < n) {
        mindistance = INFINITY;
        // Find the node with the smallest distance
        for (i = 0; i < n; i++) {
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }
        }

        // Mark the node as visited
        visited[nextnode] = 1;

        // Update the distance of adjacent nodes
        for (i = 0; i < n; i++) {
            if (!visited[i] && (mindistance + cost[nextnode][i] < distance[i])) {
                distance[i] = mindistance + cost[nextnode][i];
                pred[i] = nextnode;
            }
        }

        count++;
    }

    // Print the path and distance of each node
    for (i = 0; i < n; i++) {
        if (i != startnode) {
            printf("\nDistance of node %d = %d", i, distance[i]);
            printf("\nPath = %d", i);
            j = i;
            while (j != startnode) {
                j = pred[j];
                printf(" <- %d", j);
            }
            printf("\n");
        }
    }
}
```

OUTPUT

```
Enter number of vertices: 2

Enter the adjacency matrix:
1
2
```

3
4

Enter the starting node: 2

Distance of node 0 = 0
Path = 0 <- 2

Distance of node 1 = 0
Path = 1 <- 2

**Ex. No. 10 PRIM'S ALGORITHM**

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

#define INFINITY 9999
#define MAX 20

int G[MAX][MAX], spanning[MAX][MAX], n;

int prims();

int main() {
    int i, j, total_cost;

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

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

    total_cost = prims();

    printf("\nSpanning tree matrix:\n");
    for (i = 0; i < n; i++) {
        printf("\n");
        for (j = 0; j < n; j++) {
            printf("%d\t", spanning[i][j]);
        }
    }

    printf("\n\nTotal cost of spanning tree = %d\n", total_cost);

    return 0;
}
```

```
int prims() {
    int cost[MAX][MAX];
    int distance[MAX], from[MAX];
    int visited[MAX], no_of_edges, min_cost = 0;
    int i, j, u, v, min_distance;

    // Create the cost matrix and initialize the spanning matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (G[i][j] == 0) {
                cost[i][j] = INFINITY;
            } else {
                cost[i][j] = G[i][j];
            }
            spanning[i][j] = 0;
        }
    }

    // Initialize visited[], distance[], and from[]
    distance[0] = 0;
    visited[0] = 1;
    for (i = 1; i < n; i++) {
        distance[i] = cost[0][i];
        from[i] = 0;
        visited[i] = 0;
    }

    no_of_edges = n - 1; // Number of edges to be added

    while (no_of_edges > 0) {
        // Find the vertex at minimum distance from the tree
        min_distance = INFINITY;
        for (i = 1; i < n; i++) {
            if (!visited[i] && distance[i] < min_distance) {
                v = i;
                min_distance = distance[i];
            }
        }

        u = from[v];

        // Insert the edge into the spanning tree
        spanning[u][v] = distance[v];
        spanning[v][u] = distance[v];
        no_of_edges--;

        visited[v] = 1;

        // Update the distance[] array
        for (i = 1; i < n; i++) {
```

```
            if (!visited[i] && cost[i][v] < distance[i]) {
                distance[i] = cost[i][v];
                from[i] = v;
            }
        }

        min_cost += cost[u][v];
    }

    return min_cost;
}
```

OUTPUT
Enter number of vertices: 3

Enter the adjacency matrix:
1
2
3
4
5
6
7
8
9

Spanning tree matrix:

| | | |
|---|---|---|
| 0 | 2 | 3 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |

Total cost of spanning tree = 5

**Ex. No. 11 A LINEAR SEARCH**

```
#include <stdio.h>

int main() {
    int a[50], i, n, val, found;

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

    printf("Enter Array Elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
```

```
    printf("Enter element to locate: ");
    scanf("%d", &val);

    found = 0;
    for (i = 0; i < n; i++) {
        if (a[i] == val) {
            printf("Element found at position %d\n", i + 1); // Positions typically start at 1
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Element not found\n");
    }

    return 0;
}
```

OUTPUT
Enter the number of elements: 5
Enter Array Elements:
1
2
3
4
5
Enter element to locate: 2
Element found at position 2

**Ex. No. 11 B BINARY SEARCH**

```
#include <stdio.h>

int main() {
    int a[50], i, n, upper, lower, mid, val, found, att = 0;

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

    for (i = 0; i < n; i++)
        a[i] = 2 * i;

    printf("\nElements in Sorted Order:\n");
```

```
    for (i = 0; i < n; i++)
        printf("%4d", a[i]);

    printf("\nEnter element to locate: ");
    scanf("%d", &val);

    upper = n - 1; // Corrected: upper bound should be n-1
    lower = 0;
    found = 0;

    while (lower <= upper) {
        mid = (upper + lower) / 2;
        att++;

        if (a[mid] == val) {
            printf("Found at index %d in %d attempts\n", mid, att);
            found = 1;
            break;
        } else if (a[mid] > val) {
            upper = mid - 1;
        } else {
            lower = mid + 1;
        }
    }

    if (!found) {
        printf("Element not found\n");
    }

    return 0;
}
```

OUTPUT

```
Enter array size: 10

Elements in Sorted Order:
   0   2   4   6   8  10  12  14  16  18
Enter element to locate: 16
Found at index 8 in 3 attempts
```

**Ex. No. 12 A INSERTION SORT**

```
#include <stdio.h>

int main() {
    int i, j, k, n, temp, a[20];

    printf("Enter total elements: ");
    scanf("%d", &n);
```

```
    printf("Enter array elements: ");
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    for(i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;

        while((j >= 0) && (temp < a[j])) {
            a[j + 1] = a[j];
            j = j - 1;
        }

        a[j + 1] = temp;

        printf("\nAfter Pass %d: ", i);
        for(k = 0; k < n; k++) {
            printf("%d ", a[k]);
        }
    }

    printf("\nSorted List: ");
    for(i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }

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

OUTPUT

```
Enter total elements: 6
Enter array elements: 1
5
3
6
7
8

After Pass 1: 1 5 3 6 7 8
After Pass 2: 1 3 5 6 7 8
After Pass 3: 1 3 5 6 7 8
After Pass 4: 1 3 5 6 7 8
After Pass 5: 1 3 5 6 7 8
Sorted List: 1 3 5 6 7 8
```

Ex. No. 12 B SELECTION SORT

```
#include <stdio.h>

void selection_sort(int a[], int size) {
    int temp, i, j, min;
    for (i = 0; i < size - 1; i++) {
        min = i; // Considering element i as the minimum
        for (j = i + 1; j < size; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // Swap the found minimum element with the first element
        temp = a[min];
        a[min] = a[i];
        a[i] = temp;
    }
}

int main() {
    int arr[10], i;

    printf("Enter 10 values:\n");
    for (i = 0; i < 10; i++) {
        scanf("%d", &arr[i]);
    }

    // Call the selection sort function
    selection_sort(arr, 10);

    printf("\nSorted Values:\n");
    for (i = 0; i < 10; i++) {
        printf("%d\n", arr[i]);
    }

    return 0;
}
```

OUTPUT

```
Enter 10 values:
12
45
25
36
54
87
65
```

34
34
98

Sorted Values:
12
25
34
34
36
45
54
65
87
98

**Ex. No. 13 MERGE SORT**

```
#include <stdio.h>

void merge(int arr[], int min, int mid, int max);
void part(int arr[], int min, int max);
int size;

int main() {
   int i, arr[30];

   printf("Enter total number of elements (max 30): ");
   scanf("%d", &size);

   // Ensure the size is within bounds
   if (size > 30) {
      printf("Size exceeds the maximum allowed (30).\n");
      return 1;
   }

   printf("Enter array elements: ");
   for (i = 0; i < size; i++)
      scanf("%d", &arr[i]);

   part(arr, 0, size - 1);

   printf("\nMerge sorted list: ");
   for (i = 0; i < size; i++)
      printf("%d ", arr[i]);

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

```
void part(int arr[], int min, int max) {
    int mid;

    if (min < max) {
        mid = (min + max) / 2;
        part(arr, min, mid);
        part(arr, mid + 1, max);
        merge(arr, min, mid, max);

        // Print the half-sorted list after merging the first half
        if (max - min == (size / 2) - 1) {
            printf("\n\nHalf sorted list: ");
            for (int i = min; i <= max; i++)
                printf("%d ", arr[i]);
            printf("\n");
        }
    }
}

void merge(int arr[], int min, int mid, int max) {
    int tmp[30];
    int i, j, k, m;
    j = min;
    m = mid + 1;

    for (i = min; j <= mid && m <= max; i++) {
        if (arr[j] <= arr[m]) {
            tmp[i] = arr[j];
            j++;
        } else {
            tmp[i] = arr[m];
            m++;
        }
    }

    // Copy the remaining elements of the left subarray, if any
    if (j > mid) {
        for (k = m; k <= max; k++) {
            tmp[i] = arr[k];
            i++;
        }
    } else {
        for (k = j; k <= mid; k++) {
            tmp[i] = arr[k];
            i++;
        }
    }

    // Copy the merged elements back into the original array
    for (k = min; k <= max; k++)
        arr[k] = tmp[k];
```

}

OUTPUT

Enter total number of elements (max 30): 8
Enter array elements: 4
58

9
6
2
8
9
10

Half sorted list: 4 5 6 9

Half sorted list: 2 8 9 10

Merge sorted list: 2 4 5 6 8 9 9 10

**Experiments beyond the Syllabus**
**REPRESENTATION OF GRAPH**

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

// Initialize the matrix to 0
void init(int arr[][V]) {
    int i, j;
    for(i = 0; i < V; i++)
        for(j = 0; j < V; j++)
            arr[i][j] = 0;
}

// Add edge: set arr[src][dest] = 1
void addEdge(int arr[][V], int src, int dest) {
    arr[src][dest] = 1;
}

// Print the adjacency matrix
void printAdjMatrix(int arr[][V]) {
    int i, j;
    for(i = 0; i < V; i++) {
        for(j = 0; j < V; j++) {
            printf("%d ", arr[i][j]);
```

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

// Main function
int main() {
    int adjMatrix[V][V];

    init(adjMatrix);

    // Adding edges
    addEdge(adjMatrix, 0, 1);
    addEdge(adjMatrix, 0, 2);
    addEdge(adjMatrix, 0, 3);
    addEdge(adjMatrix, 1, 3);
    addEdge(adjMatrix, 1, 4);
    addEdge(adjMatrix, 2, 3);
    addEdge(adjMatrix, 3, 4);

    // Printing the adjacency matrix
    printAdjMatrix(adjMatrix);

    return 0;
}
```

OUTPUT

```
0 1 1 1 0
0 0 0 1 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```