Data Structures Lab Programs (Part 1)

ISL36 - III Semester

Table of Contents

Program 1: Student Structure

```
#include <stdio.h>

// Structure definition for student
struct student {
    char name[50];
    char usn[10];
    int age;
    float cgpa;
};

int main() {
    struct student s[100];  // Array of students
    int n;                  // Number of students
    float min_cgpa;         // CGPA threshold

    // Input number of students
    printf("Enter number of students: ");
    scanf("%d", &n);

    // Input student details
    for(int i = 0; i < n; i++) {
        printf("\nStudent %d details:\n", i+1);
        printf("Name: ");
        scanf("%s", s[i].name);
        printf("USN: ");
        scanf("%s", s[i].usn);
        printf("Age: ");
        scanf("%d", &s[i].age);
        printf("CGPA: ");
        scanf("%f", &s[i].cgpa);
    }

    // Input CGPA threshold
    printf("\nEnter minimum CGPA to display: ");
    scanf("%f", &min_cgpa);

    // Display students with CGPA >= threshold
    printf("\nStudents with CGPA >= %.2f:\n", min_cgpa);
    for(int i = 0; i < n; i++) {
        if(s[i].cgpa >= min_cgpa) {
            printf("\nName: %s", s[i].name);
            printf("\nUSN: %s", s[i].usn);
            printf("\nAge: %d", s[i].age);
```

```
            printf("\nCGPA: %.2f\n", s[i].cgpa);
        }
    }
    return 0;
}
```

Program 2: Recursive Programs

a) Tower of Hanoi

```
#include <stdio.h>

// Recursive function to solve Tower of Hanoi
void towerOfHanoi(int n, char from, char aux, char to) {
    if (n == 1) {
        printf("\nMove disk 1 from %c to %c", from, to);
        return;
    }
    towerOfHanoi(n-1, from, to, aux);
    printf("\nMove disk %d from %c to %c", n, from, to);
    towerOfHanoi(n-1, aux, from, to);
}

int main() {
    int n;
    printf("Enter number of disks: ");
    scanf("%d", &n);
    towerOfHanoi(n, 'A', 'B', 'C');
    return 0;
}
```

b) Fibonacci Series

Program to generate Fibonacci series using recursion

```
#include <stdio.h>

// Function to get nth Fibonacci number
int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    int n;
    printf("Enter number of terms: ");
    scanf("%d", &n);

    printf("Fibonacci Series: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", fib(i));
    }
    return 0;
}
```

Program 3: Stack Operations

Implementation of stack with additional palindrome checking functionality

```
#include <stdio.h>
#include <string.h>
#define MAX 50

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

// Function prototypes
void push(int);
int pop(void);
void display(void);
int isPalindrome(char[]);

// Push element to stack
void push(int x) {
    if(top >= MAX-1) {
        printf("\nStack Overflow");
        return;
    }
    stack[++top] = x;
}

// Pop element from stack
int pop() {
    if(top < 0) {
        printf("\nStack Underflow");
        return -1;
    }
    return stack[top--];
}

// Display stack elements
void display() {
    if(top < 0) {
        printf("\nStack Empty");
        return;
    }
    printf("\nStack elements: ");
    for(int i = 0; i <= top; i++)
        printf("%d ", stack[i]);
}
```

```
// Check if string is palindrome using stack
int isPalindrome(char str[]) {
    int length = strlen(str);
    int i;

    // Push first half of string
    for(i = 0; i < length/2; i++)
        push(str[i]);

    // Skip middle character for odd length
    if(length % 2 != 0)
        i++;

    // Compare remaining characters
    while(str[i] != '\0') {
        if(str[i] != pop())
            return 0;
        i++;
    }
    return 1;
}

int main() {
    int choice, x;
    char str[MAX];

    while(1) {
        printf("\n\nStack Operations:");
        printf("\n1. Push");
        printf("\n2. Pop");
        printf("\n3. Display");
        printf("\n4. Check Palindrome");
        printf("\n5. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                push(x);
                break;
            case 2:
                x = pop();
```

```
            if(x != -1)
                printf("\nPopped: %d", x);
            break;
        case 3:
            display();
            break;
        case 4:
            printf("\nEnter string: ");
            scanf("%s", str);
            if(isPalindrome(str))
                printf("\nPalindrome");
            else
                printf("\nNot Palindrome");
            break;
        case 5:
            return 0;
    }
  }
}
```

Program 4: Infix to Postfix Conversion

Program to convert infix expression to postfix using stack

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100

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

// Function prototypes
void push(char);
char pop(void);
int precedence(char);

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

// Pop operator from stack
char pop() {
    if(top == -1)
        return -1;
    return stack[top--];
}

// Get operator precedence
int precedence(char c) {
    switch(c) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return -1;
}

int main() {
    char infix[MAX], postfix[MAX], ch;
    int i, j = 0;
```

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

    printf("Postfix expression: ");

    for(i = 0; infix[i]; i++) {
        // If operand, add to postfix
        if(isalnum(infix[i]))
            postfix[j++] = infix[i];

        // If opening bracket, push to stack
        else if(infix[i] == '(')
            push(infix[i]);

        // If closing bracket, pop until opening bracket
        else if(infix[i] == ')') {
            while((ch = pop()) != '(')
                postfix[j++] = ch;
        }

        // If operator
        else {
            while(top != -1 && precedence(stack[top]) >= precedence(infix[i]))
                postfix[j++] = pop();
            push(infix[i]);
        }
    }

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

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

    return 0;
}
```

Program 5: Postfix Evaluation

Program to evaluate postfix expression using stack

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100

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

// Function prototypes
void push(int);
int pop(void);

// Push operand to stack
void push(int x) {
    stack[++top] = x;
}

// Pop operand from stack
int pop() {
    return stack[top--];
}

int main() {
    char exp[MAX];
    int i = 0, op1, op2, val;

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

    while(exp[i] != '\0') {
        // If operand, push to stack
        if(isdigit(exp[i]))
            push(exp[i] - '0');

        // If operator, pop operands and evaluate
        else {
            op2 = pop();
            op1 = pop();
            switch(exp[i]) {
                case '+': val = op1 + op2; break;
                case '-': val = op1 - op2; break;
                case '*': val = op1 * op2; break;
                case '/': val = op1 / op2; break;
```

```
            }
            push(val);
        }
        i++;
    }

    printf("Result: %d", pop());
    return 0;
}
```

Program 6: Process Scheduling Queue

Implementation of process scheduling using queue

```
#include <stdio.h>
#define MAX 50

// Global variables
int queue[MAX];
int front = -1, rear = -1;

// Function prototypes
void enqueue(int);
int dequeue(void);
void display(void);

// Add process to queue
void enqueue(int pid) {
    if(rear == MAX-1) {
        printf("\nQueue Full");
        return;
    }
    if(front == -1)
        front = 0;
    queue[++rear] = pid;
    printf("\nProcess %d added to queue", pid);
}

// Remove process from queue
int dequeue() {
    if(front == -1 || front > rear) {
        printf("\nQueue Empty");
        return -1;
    }
    int pid = queue[front++];
    printf("\nProcess %d removed from queue", pid);
    return pid;
}

// Display queue contents
void display() {
    if(front == -1 || front > rear) {
        printf("\nQueue Empty");
        return;
    }
    printf("\nProcesses in queue: ");
    for(int i = front; i <= rear; i++)
```

```
        printf("%d ", queue[i]);
}

int main() {
    int choice, pid;

    while(1) {
        printf("\n\nProcess Queue Operations:");
        printf("\n1. Add Process");
        printf("\n2. Remove Process");
        printf("\n3. Display Queue");
        printf("\n4. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter Process ID: ");
                scanf("%d", &pid);
                enqueue(pid);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
        }
    }
}
```

Program 7: Circular Queue

Implementation of circular queue for character data

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

// Global variables
char queue[MAX];
int front = -1, rear = -1;

// Function prototypes
int isFull(void);
int isEmpty(void);
void insert(char);
char delete(void);
void display(void);

// Check if queue is full
int isFull() {
    return (rear + 1) % MAX == front;
}

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

// Insert element into queue
void insert(char c) {
    if(isFull()) {
        printf("\nQueue Full");
        return;
    }
    if(isEmpty())
        front = 0;
    rear = (rear + 1) % MAX;
    queue[rear] = c;
    printf("\nInserted: %c", c);
}

// Delete element from queue
char delete() {
    if(isEmpty()) {
        printf("\nQueue Empty");
        return '\0';
    }
```

```
    char c = queue[front];
    if(front == rear)
        front = rear = -1;
    else
        front = (front + 1) % MAX;
    printf("\nDeleted: %c", c);
    return c;
}

// Display queue contents
void display() {
    if(isEmpty()) {
        printf("\nQueue Empty");
        return;
    }
    printf("\nQueue elements: ");
    int i = front;
    do {
        printf("%c ", queue[i]);
        i = (i + 1) % MAX;
    } while(i != (rear + 1) % MAX);
}

int main() {
    int choice;
    char c;

    while(1) {
        printf("\n\nCircular Queue Operations:");
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Display");
        printf("\n4. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);
        getchar();  // Clear input buffer

        switch(choice) {
            case 1:
                printf("\nEnter character: ");
                scanf("%c", &c);
                insert(c);
                break;
            case 2:
```

```
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
        }
    }
}
```

Program 8: Singly Linked List

Implementation of singly linked list with basic operations

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

// Node structure definition
struct Node {
    int data;
    struct Node* next;
};

// Global head pointer
struct Node* head = NULL;

// Function prototypes
void insertFront(int);
void deleteRear(void);
void search(int);
void display(void);

// Insert node at front
void insertFront(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = head;
    head = newNode;
    printf("\nInserted %d at front", x);
}

// Delete node from rear
void deleteRear() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    // If only one node
    if(head->next == NULL) {
        free(head);
        head = NULL;
        printf("\nDeleted from rear");
        return;
    }

    // Traverse to second last node
```

```
    struct Node* temp = head;
    while(temp->next->next != NULL)
        temp = temp->next;

    free(temp->next);
    temp->next = NULL;
    printf("\nDeleted from rear");
}

// Search for an element
void search(int x) {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    struct Node* temp = head;
    int position = 1;

    while(temp != NULL) {
        if(temp->data == x) {
            printf("\nElement %d found at position %d", x, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("\nElement %d not found", x);
}

// Display list contents
void display() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    printf("\nList elements: ");
    struct Node* temp = head;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

```c
int main() {
    int choice, x;

    while(1) {
        printf("\n\nSingly Linked List Operations:");
        printf("\n1. Insert at Front");
        printf("\n2. Delete from Rear");
        printf("\n3. Search");
        printf("\n4. Display");
        printf("\n5. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                insertFront(x);
                break;
            case 2:
                deleteRear();
                break;
            case 3:
                printf("\nEnter element to search: ");
                scanf("%d", &x);
                search(x);
                break;
            case 4:
                display();
                break;
            case 5:
                return 0;
        }
    }
}
```

Program 9: Doubly Linked List

Implementation of doubly linked list with basic operations

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

// Node structure definition
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Global head pointer
struct Node* head = NULL;

// Function prototypes
void insertEnd(int);
void insertFront(int);
void deleteFront(void);
void search(int);
void display(void);

// Insert at end of list
void insertEnd(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = NULL;

    if(head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        printf("\nInserted %d at end", x);
        return;
    }

    struct Node* temp = head;
    while(temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
    newNode->prev = temp;
    printf("\nInserted %d at end", x);
}

// Insert at front of list
```

```c
void insertFront(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->prev = NULL;
    newNode->next = head;

    if(head != NULL)
        head->prev = newNode;

    head = newNode;
    printf("\nInserted %d at front", x);
}

// Delete from front
void deleteFront() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    struct Node* temp = head;
    head = head->next;

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

    free(temp);
    printf("\nDeleted from front");
}

// Search for an element
void search(int x) {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    struct Node* temp = head;
    int position = 1;

    while(temp != NULL) {
        if(temp->data == x) {
            printf("\nElement %d found at position %d", x, position);
            return;
```

```
        }
        temp = temp->next;
        position++;
    }
    printf("\nElement %d not found", x);
}

// Display list contents
void display() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    printf("\nList elements: ");
    struct Node* temp = head;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    int choice, x;

    while(1) {
        printf("\n\nDoubly Linked List Operations:");
        printf("\n1. Insert at End");
        printf("\n2. Insert at Front");
        printf("\n3. Delete from Front");
        printf("\n4. Search");
        printf("\n5. Display");
        printf("\n6. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                insertEnd(x);
                break;
            case 2:
                printf("\nEnter element: ");
```

```
                scanf("%d", &x);
                insertFront(x);
                break;
            case 3:
                deleteFront();
                break;
            case 4:
                printf("\nEnter element to search: ");
                scanf("%d", &x);
                search(x);
                break;
            case 5:
                display();
                break;
            case 6:
                return 0;
        }
    }
}
```

Program 10: Circular Linked List

Implementation of circular linked list with basic operations

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

// Node structure definition
struct Node {
    int data;
    struct Node* next;
};

// Global head pointer
struct Node* head = NULL;

// Function prototypes
void insertFront(int);
void insertEnd(int);
void deleteEnd(void);
void search(int);
void display(void);

// Insert at front of list
void insertFront(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;

    if(head == NULL) {
        head = newNode;
        newNode->next = head;
        printf("\nInserted %d at front", x);
        return;
    }

    struct Node* temp = head;
    while(temp->next != head)
        temp = temp->next;

    newNode->next = head;
    temp->next = newNode;
    head = newNode;
    printf("\nInserted %d at front", x);
}

// Insert at end of list
void insertEnd(int x) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;

    if(head == NULL) {
        head = newNode;
        newNode->next = head;
        printf("\nInserted %d at end", x);
        return;
    }

    struct Node* temp = head;
    while(temp->next != head)
        temp = temp->next;

    temp->next = newNode;
    newNode->next = head;
    printf("\nInserted %d at end", x);
}

// Delete from end
void deleteEnd() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    if(head->next == head) {
        free(head);
        head = NULL;
        printf("\nDeleted from end");
        return;
    }

    struct Node* temp = head;
    while(temp->next->next != head)
        temp = temp->next;

    free(temp->next);
    temp->next = head;
    printf("\nDeleted from end");
}

// Search for an element
void search(int x) {
```

```
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    struct Node* temp = head;
    int position = 1;

    do {
        if(temp->data == x) {
            printf("\nElement %d found at position %d", x, position);
            return;
        }
        temp = temp->next;
        position++;
    } while(temp != head);

    printf("\nElement %d not found", x);
}

// Display list contents
void display() {
    if(head == NULL) {
        printf("\nList Empty");
        return;
    }

    struct Node* temp = head;
    printf("\nList elements: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while(temp != head);
}

int main() {
    int choice, x;

    while(1) {
        printf("\n\nCircular Linked List Operations:");
        printf("\n1. Insert at Front");
        printf("\n2. Insert at End");
        printf("\n3. Delete from End");
        printf("\n4. Search");
```

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

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                insertFront(x);
                break;
            case 2:
                printf("\nEnter element: ");
                scanf("%d", &x);
                insertEnd(x);
                break;
            case 3:
                deleteEnd();
                break;
            case 4:
                printf("\nEnter element to search: ");
                scanf("%d", &x);
                search(x);
                break;
            case 5:
                display();
                break;
            case 6:
                return 0;
        }
    }
}
```

Program 11: Stack using Linked List

Implementation of stack using linked list

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

// Node structure definition
struct Node {
    int data;
    struct Node* next;
};

// Global top pointer
struct Node* top = NULL;

// Function prototypes
void push(int);
void pop(void);
void display(void);

// Push element onto stack
void push(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = top;
    top = newNode;
    printf("\nPushed %d onto stack", x);
}

// Pop element from stack
void pop() {
    if(top == NULL) {
        printf("\nStack Underflow");
        return;
    }

    struct Node* temp = top;
    printf("\nPopped %d from stack", temp->data);
    top = top->next;
    free(temp);
}

// Display stack contents
void display() {
    if(top == NULL) {
        printf("\nStack Empty");
```

```
        return;
    }

    printf("\nStack elements: ");
    struct Node* temp = top;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    int choice, x;

    while(1) {
        printf("\n\nStack Operations:");
        printf("\n1. Push");
        printf("\n2. Pop");
        printf("\n3. Display");
        printf("\n4. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                push(x);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
        }
    }
}
```

Program 12: Queue using Linked List

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

// Node structure definition
struct Node {
    int data;
    struct Node* next;
};

// Global front and rear pointers
struct Node *front = NULL, *rear = NULL;

// Function prototypes
void enqueue(int);
void dequeue(void);
void display(void);

// Add element to queue
void enqueue(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = NULL;

    if(rear == NULL) {
        front = rear = newNode;
        printf("\nEnqueued %d", x);
        return;
    }

    rear->next = newNode;
    rear = newNode;
    printf("\nEnqueued %d", x);
}

// Remove element from queue
void dequeue() {
    if(front == NULL) {
        printf("\nQueue Empty");
        return;
    }

    struct Node* temp = front;
    printf("\nDequeued %d", temp->data);
```

```
    front = front->next;

    if(front == NULL)
        rear = NULL;

    free(temp);
}

// Display queue contents
void display() {
    if(front == NULL) {
        printf("\nQueue Empty");
        return;
    }

    printf("\nQueue elements: ");
    struct Node* temp = front;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int main() {
    int choice, x;

    while(1) {
        printf("\n\nQueue Operations:");
        printf("\n1. Enqueue");
        printf("\n2. Dequeue");
        printf("\n3. Display");
        printf("\n4. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter element: ");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
```

```
            case 3:
                display();
                break;
            case 4:
                return 0;
        }
    }
}
```

Program 13: Binary Search Tree

Implementation of Binary Search Tree for student score management

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

// Node structure definition
struct Node {
    int score;
    struct Node* left;
    struct Node* right;
};

// Global root pointer
struct Node* root = NULL;

// Function prototypes
struct Node* createNode(int);
struct Node* insert(struct Node*, int);
void inorder(struct Node*);
int findMin(struct Node*);
int findMax(struct Node*);
void search(struct Node*, int);

// Create a new node
struct Node* createNode(int score) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->score = score;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Insert a score into BST
struct Node* insert(struct Node* node, int score) {
    if(node == NULL) {
        printf("\nScore %d inserted", score);
        return createNode(score);
    }

    if(score < node->score)
        node->left = insert(node->left, score);
    else if(score > node->score)
        node->right = insert(node->right, score);
    else
        printf("\nDuplicate score not allowed");
```

```
    return node;
}

// Display scores in ascending order (inorder traversal)
void inorder(struct Node* node) {
    if(node != NULL) {
        inorder(node->left);
        printf("%d ", node->score);
        inorder(node->right);
    }
}

// Find minimum score
int findMin(struct Node* node) {
    if(node == NULL) {
        printf("\nTree Empty");
        return -1;
    }

    while(node->left != NULL)
        node = node->left;
    return node->score;
}

// Find maximum score
int findMax(struct Node* node) {
    if(node == NULL) {
        printf("\nTree Empty");
        return -1;
    }

    while(node->right != NULL)
        node = node->right;
    return node->score;
}

// Search for a score
void search(struct Node* node, int score) {
    if(node == NULL) {
        printf("\nScore %d not found", score);
        return;
    }

    if(node->score == score) {
```

```
        printf("\nScore %d found in the BST", score);
        return;
    }

    if(score < node->score)
        search(node->left, score);
    else
        search(node->right, score);
}

int main() {
    int choice, score;

    while(1) {
        printf("\n\nBinary Search Tree Operations:");
        printf("\n1. Insert Score");
        printf("\n2. Display Scores (Ascending)");
        printf("\n3. Find Highest and Lowest Scores");
        printf("\n4. Search Score");
        printf("\n5. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter score (1-100): ");
                scanf("%d", &score);
                if(score < 1 || score > 100)
                    printf("\nInvalid score! Score should be between 1 and 100");
                else
                    root = insert(root, score);
                break;
            case 2:
                printf("\nScores in ascending order: ");
                if(root == NULL)
                    printf("Tree Empty");
                else
                    inorder(root);
                break;
            case 3:
                score = findMax(root);
                if(score != -1)
                    printf("\nHighest score: %d", score);
                score = findMin(root);
```

```
                if(score != -1)
                    printf("\nLowest score: %d", score);
                break;
            case 4:
                printf("\nEnter score to search: ");
                scanf("%d", &score);
                search(root, score);
                break;
            case 5:
                return 0;
        }
    }
}
```