1(a) Decimal to Binary using Recursion

```
#include <stdio.h>
void convert(int n) {
    if (n == 0) return;
    convert(n / 2);
    printf("%d", n % 2);
}
int main() {
    int n;
    scanf("%d", &n);
    if (n == 0) printf("0");
    else convert(n);
    return 0;
}
```

1(b) Pyramid Pattern using Recursion

```
#include <stdio.h>
void printSpace(int s) {
    if (s == 0) return;
    printf(" ");
    printSpace(s - 1);
}
void printStar(int s) {
    if (s == 0) return;
    printf("* ");
    printStar(s - 1);
}
void pyramid(int r, int i) {
    if (i > r) return;
    printSpace(r - i);
    printStar(i);
    printf("\n");
    pyramid(r, i + 1);
}
int main() {
    int n;
    scanf("%d", &n);
    pyramid(n, 1);
    return 0;
}
```

1(c) Tower of Hanoi using Recursion

```
#include <stdio.h>
void hanoi(int n, char A, char B, char C) {
    if (n == 0) return;
    hanoi(n - 1, A, C, B);
    printf("%c to %c\n", A, C);
    hanoi(n - 1, B, A, C);
}
int main() {
    int n;
    scanf("%d", &n);
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

2.IMPLEMENTATION OF ARRAY AND ITS OPERATIONS

```
#include <stdio.h>

int main() {

    int a[100], n, i, ch, val, pos, f = 0;

    scanf("%d", &n);

    for(i = 0; i < n; i++) scanf("%d", &a[i]);

    do {

        scanf("%d", &ch);

        if(ch == 1) {

            scanf("%d %d", &pos, &val);

            if(pos >= 0 && pos <= n) {

                for(i = n; i > pos; i--) a[i] = a[i - 1];

                a[pos] = val;

                n++;

            }
```

```
} else if(ch == 2) {
    scanf("%d", &pos);
    if(pos >= 0 && pos < n) {
        for(i = pos; i < n - 1; i++) a[i] = a[i + 1];
        n--;
    }
} else if(ch == 3) {
    scanf("%d", &val);
    f = 0;
    for(i = 0; i < n; i++) {
        if(a[i] == val) {
            printf("%d\n", i);
            f = 1;
            break;
        }
    }
    if(f == 0) printf("-1\n");
} else if(ch == 4) {
    scanf("%d %d", &pos, &val);
    if(pos >= 0 && pos < n) a[pos] = val;
} else if(ch == 5) {
    for(i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
}
```

```
    } while(ch != 0);

    return 0;

}
```

3(a).Stack operations

```
#include <stdio.h>
#define SIZE 100
int main() {
    int stack[SIZE], top = -1, ch, val, i;
    do {
        scanf("%d", &ch);
        if(ch == 1) {
            if(top == SIZE - 1) printf("Stack Full\n");
            else {
                scanf("%d", &val);
                stack[++top] = val;
            }
        } else if(ch == 2) {
            if(top == -1) printf("Stack Empty\n");
            else top--;
        } else if(ch == 3) {
            if(top == -1) printf("Stack Empty\n");
            else printf("%d\n", stack[top]);
        } else if(ch == 4) {
            if(top == -1) printf("Stack Empty\n");
            else for(i = top; i >= 0; i--) printf("%d ", stack[i]);
            if(top != -1) printf("\n");
        } else if(ch == 5) {
            if(top == -1) printf("Yes\n");
            else printf("No\n");
        } else if(ch == 6) {
            if(top == SIZE - 1) printf("Yes\n");
            else printf("No\n");
        }
    } while(ch != 0);
    return 0;
}
```

3(b).Infix to postfix conversion

```
#include <stdio.h>
#include <string.h>
#define SIZE 100

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

int prec(char op) {
    if(op == '+' || op == '-') return 1;
    if(op == '*' || op == '/') return 2;
    return 0;
}

void push(char ch) {
    stack[++top] = ch;
}

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

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

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

int main() {
    char infix[SIZE], postfix[SIZE];
    int i = 0, j = 0;
    scanf("%s", infix);
    while(infix[i] != '\0') {
        char ch = infix[i];
        if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9')) {
            postfix[j++] = ch;
        } else if(ch == '(') {
            push(ch);
        } else if(ch == ')') {
            while(!isEmpty() && peek() != '(') {
```

```
            postfix[j++] = pop();
        }
        pop();
    } else {
        while(!isEmpty() && prec(peek()) >= prec(ch)) {
            postfix[j++] = pop();
        }
        push(ch);
    }
    i++;
}
while(!isEmpty()) {
    postfix[j++] = pop();
}
postfix[j] = '\0';
printf("%s\n", postfix);
return 0;
}
```

3(c).Postfix cal

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

int stack[SIZE], top = -1;

void push(int n) {
    stack[++top] = n;
}

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

int main() {
    char exp[SIZE];
    int i, a, b;
    scanf("%s", exp);
    for(i = 0; exp[i] != '\0'; i++) {
        if(isdigit(exp[i])) {
            push(exp[i] - '0');
        } else {
```

```
            a = pop();
            b = pop();
            if(exp[i] == '+') push(b + a);
            else if(exp[i] == '-') push(b - a);
            else if(exp[i] == '*') push(b * a);
            else if(exp[i] == '/') push(b / a);
        }
    }
    printf("%d\n", pop());
    return 0;
}
```

4(a).Queue operations

```
#include <stdio.h>
#define SIZE 100

int main() {
    int q[SIZE], front = -1, rear = -1, ch, val, i, f = 0;
    do {
        scanf("%d", &ch);
        if(ch == 1) {
            if(rear == SIZE - 1) printf("Queue Full\n");
            else {
                scanf("%d", &val);
                if(front == -1) front = 0;
                q[++rear] = val;
            }
        } else if(ch == 2) {
            if(front == -1 || front > rear) printf("Queue Empty\n");
            else front++;
        } else if(ch == 3) {
            if(front == -1 || front > rear) printf("Queue Empty\n");
            else {
                for(i = front; i <= rear; i++) printf("%d ", q[i]);
                printf("\n");
            }
        } else if(ch == 4) {
            scanf("%d", &val);
            f = 0;
            for(i = front; i <= rear; i++) {
                if(q[i] == val) {
                    printf("%d\n", i - front);
```

```
                f = 1;
                break;
            }
        }
        if(f == 0) printf("-1\n");
    } else if(ch == 5) {
        if(rear == SIZE - 1) printf("Yes\n");
        else printf("No\n");
    } else if(ch == 6) {
        if(front == -1 || front > rear) printf("Yes\n");
        else printf("No\n");
    }
  } while(ch != 0);
  return 0;
}
```

4(b).FCFS

```
#include <stdio.h>

int main() {
  int n, i;
  int bt[100], at[100], wt[100], tat[100], ct[100];
  int total_wt = 0, total_tat = 0;

  scanf("%d", &n);
  for(i = 0; i < n; i++) {
    scanf("%d", &at[i]);
  }
  for(i = 0; i < n; i++) {
    scanf("%d", &bt[i]);
  }

  ct[0] = at[0] + bt[0];
  tat[0] = ct[0] - at[0];
  wt[0] = tat[0] - bt[0];

  for(i = 1; i < n; i++) {
    if(ct[i - 1] < at[i]) ct[i] = at[i] + bt[i];
    else ct[i] = ct[i - 1] + bt[i];
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
  }
```

```
    for(i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    for(i = 0; i < n; i++) {
        printf("%d %d %d %d %d\n", i + 1, at[i], bt[i], wt[i], tat[i]);
    }

    printf("%d\n", total_wt / n);
    printf("%d\n", total_tat / n);

    return 0;
}
```

5.Singly

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

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

struct node *head = NULL;

void insert_begin(int val) {
    struct node *new = malloc(sizeof(struct node));
    new->data = val;
    new->next = head;
    head = new;
}

void insert_end(int val) {
    struct node *new = malloc(sizeof(struct node));
    new->data = val;
    new->next = NULL;
    if (head == NULL)
        head = new;
    else {
        struct node *temp = head;
        while (temp->next != NULL)
            temp = temp->next;
```

```
        temp->next = new;
    }
}

void insert_after(int key, int val) {
    struct node *temp = head;
    while (temp != NULL && temp->data != key)
        temp = temp->next;
    if (temp != NULL) {
        struct node *new = malloc(sizeof(struct node));
        new->data = val;
        new->next = temp->next;
        temp->next = new;
    }
}

void delete_begin() {
    if (head != NULL) {
        struct node *temp = head;
        head = head->next;
        free(temp);
    }
}

void delete_end() {
    if (head != NULL) {
        if (head->next == NULL) {
            free(head);
            head = NULL;
        } else {
            struct node *temp = head;
            while (temp->next->next != NULL)
                temp = temp->next;
            free(temp->next);
            temp->next = NULL;
        }
    }
}

void delete_value(int val) {
    if (head != NULL) {
        if (head->data == val) {
            struct node *temp = head;
            head = head->next;
```

```c
            free(temp);
        } else {
            struct node *temp = head;
            while (temp->next != NULL && temp->next->data != val)
                temp = temp->next;
            if (temp->next != NULL) {
                struct node *del = temp->next;
                temp->next = del->next;
                free(del);
            }
        }
    }
}

void display() {
    struct node *temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

void search(int val) {
    struct node *temp = head;
    while (temp != NULL) {
        if (temp->data == val) {
            printf("Found\n");
            return;
        }
        temp = temp->next;
    }
    printf("Not Found\n");
}

int main() {
    insert_end(10);
    insert_end(20);
    insert_begin(5);
    insert_after(10, 15);
    display();
    search(15);
    delete_value(10);
    delete_begin();
```

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

6.Doubly

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

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

struct node *head = NULL;

// Insert at the beginning
void insert_begin(int val) {
    struct node *new = malloc(sizeof(struct node));
    new->data = val;
    new->prev = NULL;
    new->next = head;

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

    head = new;
}

// Insert at the end
void insert_end(int val) {
    struct node *new = malloc(sizeof(struct node));
    new->data = val;
    new->next = NULL;

    if (head == NULL) {
        new->prev = NULL;
        head = new;
        return;
    }

    struct node *temp = head;
```

```
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = new;
    new->prev = temp;
}

// Insert after a node with given key
void insert_after(int key, int val) {
    struct node *temp = head;
    while (temp != NULL && temp->data != key)
        temp = temp->next;

    if (temp == NULL) {
        printf("Node with value %d not found.\n", key);
        return;
    }

    struct node *new = malloc(sizeof(struct node));
    new->data = val;
    new->next = temp->next;
    new->prev = temp;

    if (temp->next != NULL)
        temp->next->prev = new;

    temp->next = new;
}

// Delete from beginning
void delete_begin() {
    if (head == NULL) return;

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

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

    free(temp);
}

// Delete from end
void delete_end() {
```

```
    if (head == NULL) return;

    struct node *temp = head;
    if (temp->next == NULL) {
        free(temp);
        head = NULL;
        return;
    }

    while (temp->next != NULL)
        temp = temp->next;

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

// Delete a specific node by value
void delete_value(int val) {
    struct node *temp = head;
    while (temp != NULL && temp->data != val)
        temp = temp->next;

    if (temp == NULL) return;

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

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

    free(temp);
}

// Display from head to tail
void display_forward() {
    struct node *temp = head;
    printf("Forward: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
```

```
}

// Display from tail to head
void display_reverse() {
    struct node *temp = head;
    if (temp == NULL) {
        printf("Reverse: List is empty\n");
        return;
    }

    while (temp->next != NULL)
        temp = temp->next;

    printf("Reverse: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}

// Main function for menu-driven program
int main() {
    insert_end(10);
    insert_end(20);
    insert_begin(5);
    insert_after(10, 15);

    display_forward();
    display_reverse();

    delete_value(10);
    delete_begin();
    delete_end();

    display_forward();
    return 0;
}
```

7.Single and circular

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

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

struct Node *head = NULL;

// Insert at beginning
void insert_begin(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head)
            temp = temp->next;

        newNode->next = head;
        temp->next = newNode;
        head = newNode;
    }
}

// Insert at end
void insert_end(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head)
            temp = temp->next;

        temp->next = newNode;
        newNode->next = head;
    }
}
```

```
// Delete from beginning
void delete_begin() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* last = head;
        while (last->next != head)
            last = last->next;

        head = head->next;
        last->next = head;
        free(temp);
    }
}

// Delete from end
void delete_end() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head, *prev = NULL;
    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head;
        free(temp);
    }
}

// Display the list
```

```
void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

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

// Main function with sample calls
int main() {
    insert_end(10);
    insert_end(20);
    insert_begin(5);
    display();

    delete_begin();
    display();

    delete_end();
    display();

    return 0;
}
```

8.Doubly and circular

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

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

struct Node* head = NULL;
```

```
// Insert at beginning
void insert_begin(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;

    if (head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        head = newNode;
    } else {
        struct Node* last = head->prev;

        newNode->next = head;
        newNode->prev = last;

        last->next = newNode;
        head->prev = newNode;

        head = newNode;
    }
}

// Insert at end
void insert_end(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;

    if (head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        head = newNode;
    } else {
        struct Node* last = head->prev;

        newNode->next = head;
        newNode->prev = last;

        last->next = newNode;
        head->prev = newNode;
    }
}

// Delete from beginning
void delete_begin() {
```

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

    struct Node* temp = head;

    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* last = head->prev;

        head = head->next;
        last->next = head;
        head->prev = last;

        free(temp);
    }
}

// Delete from end
void delete_end() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* last = head->prev;

    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* second_last = last->prev;

        second_last->next = head;
        head->prev = second_last;

        free(last);
    }
}

// Display forward
```

```
void display_forward() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

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

// Display backward
void display_backward() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head->prev;
    printf("Backward: ");
    struct Node* last = temp;

    do {
        printf("%d ", temp->data);
        temp = temp->prev;
    } while (temp != last);
    printf("\n");
}

// Main function with sample operations
int main() {
    insert_end(10);
    insert_end(20);
    insert_begin(5);
    display_forward();
    display_backward();

    delete_begin();
    display_forward();
    display_backward();
```

```
    delete_end();
    display_forward();
    display_backward();

    return 0;
}
```

10(a).Liner search

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;  // Return index if found
    }
    return -1; // Not found
}

int main() {
    int arr[100], n, key, pos;

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

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

    printf("Enter key to search: ");
    scanf("%d", &key);

    pos = linearSearch(arr, n, key);

    if (pos == -1)
        printf("Element not found.\n");
    else
        printf("Element found at index %d.\n", pos);

    return 0;
}
```

10(b).Binary search

```
#include <stdio.h>

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1, mid;

    while (low <= high) {
        mid = (low + high) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }

    return -1; // Not found
}

int main() {
    int arr[100], n, key, pos;

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

    printf("Enter %d sorted elements:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Enter key to search: ");
    scanf("%d", &key);

    pos = binarySearch(arr, n, key);

    if (pos == -1)
        printf("Element not found.\n");
    else
        printf("Element found at index %d.\n", pos);

    return 0;
}
```

11.Insertion sort

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

int main() {
    int arr[100], n;

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

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

    insertionSort(arr, n);

    printf("Sorted array:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

12.Hash table

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

```c
int hashTable[SIZE];

void insert(int key) {
    int index = key % SIZE;
    int i = 0;

    while (hashTable[(index + i) % SIZE] != -1) {
        i++;
        if (i == SIZE) {
            printf("Hash Table is full!\n");
            return;
        }
    }
    hashTable[(index + i) % SIZE] = key;
}

void display() {
    printf("Hash Table:\n");
    for (int i = 0; i < SIZE; i++)
        printf("Index %d: %d\n", i, hashTable[i]);
}

int main() {
    int n, key;

    // Initialize all elements to -1 (indicates empty)
    for (int i = 0; i < SIZE; i++)
        hashTable[i] = -1;

    printf("Enter number of keys to insert: ");
    scanf("%d", &n);

    printf("Enter %d keys:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &key);
        insert(key);
    }

    display();

    return 0;
}
```

BST

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

// Structure of each node
struct Node {
    int data;
    struct Node *left, *right;
};

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

// INSERT: insert node in correct BST position
struct Node* insert(struct Node* root, int value) {
    if (root == NULL)
        return createNode(value); // New tree or leaf spot found

    if (value < root->data)
        root->left = insert(root->left, value); // Go left
    else if (value > root->data)
        root->right = insert(root->right, value); // Go right

    return root;
}

// SEARCH: find value in BST
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// FIND MIN
```

```
struct Node* findMin(struct Node* root) {
    while (root && root->left != NULL)
        root = root->left;
    return root;
}

// FIND MAX
struct Node* findMax(struct Node* root) {
    while (root && root->right != NULL)
        root = root->right;
    return root;
}

// DELETE node
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) return NULL;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Case 1 & 2: 0 or 1 child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Case 3: Two children
        struct Node* temp = findMin(root->right);
        root->data = temp->data; // Copy min from right
        root->right = deleteNode(root->right, temp->data); // Delete duplicate
    }

    return root;
}

// INORDER (L → Root → R)
```

```
void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// PREORDER (Root → L → R)
void preorder(struct Node* root) {
    if (root) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// POSTORDER (L → R → Root)
void postorder(struct Node* root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

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

    // Insert values
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder: ");
    inorder(root);
    printf("\n");

    printf("Preorder: ");
    preorder(root);
```

```
    printf("\n");

    printf("Postorder: ");
    postorder(root);
    printf("\n");

    struct Node* found = search(root, 40);
    printf("Search 40: %s\n", (found ? "Found" : "Not Found"));

    printf("Min: %d\n", findMin(root)->data);
    printf("Max: %d\n", findMax(root)->data);

    root = deleteNode(root, 50); // Delete root node

    printf("Inorder after deletion: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

AVL

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

// Node structure
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
} Node;

// Get max of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Get height of tree
int height(Node* N) {
    if (N == NULL) return 0;
    return N->height;
}
```

```
// Create new node
Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    node->height = 1; // new node is leaf so height = 1
    return node;
}

// Right rotate subtree rooted with y
Node* rightRotate(Node* y) {
    Node* x = y->left;
    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 x; // new root
}

// Left rotate subtree rooted with x
Node* leftRotate(Node* x) {
    Node* y = x->right;
    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 y; // new root
}

// Get balance factor of node N
int getBalance(Node* N) {
```

```
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

// Insert a node into AVL tree
Node* insert(Node* node, int key) {
    // 1. Normal BST insert
    if (node == NULL) return newNode(key);

    if (key < node->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);
    else
        return node; // duplicate keys not allowed

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

    // 3. Get balance factor
    int balance = getBalance(node);

    // 4. If node is unbalanced, fix it

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

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

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

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

```
    // Return unchanged node pointer
    return node;
}

// Find node with minimum value
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

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

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

            if (temp == NULL) {
                // No child case
                temp = root;
                root = NULL;
            } else {
                // One child case
                *root = *temp; // Copy contents of child
            }
            free(temp);
        } else {
            // Node with two children: Get inorder successor (smallest in right subtree)
            Node* temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }

    // If tree had only one node
```

```
    if (root == NULL)
        return root;

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

    // 3. Get balance factor
    int balance = getBalance(root);

    // 4. Balance the tree

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

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

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

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

    return root;
}

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

// Preorder traversal (root, left, right)
```

```
void preorder(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

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

// Main function to test AVL tree
int main() {
    Node* root = NULL;

    // Insert nodes
    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("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    // Delete a node
    root = deleteNode(root, 40);
    printf("After deleting 40, inorder traversal: ");
```

```
    inorder(root);
    printf("\n");

    return 0;
}
```