Aligarh College Of Engineering And Technology

Approved by AICTE

AFFILIATED TO : Dr. APJ Abdul Kalam Technical University,
Lucknow, Uttar Pradesh(AKTU)

Academic Year : 2024-25

Data Structures And Algorithm Report

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE

SUBMITTED BY : MANISH TOMAR

Under the supervision of

Mr. Devendra Sharma

Department of computer science & Engineering

Head Of Department(HOD)

Dr. Anand Sharma

Submitted To

Mr. Devendra Sharma

1. Array Traversal

```c
#include <stdio.h>
int main() {
int arr[] = {10, 20, 30, 40, 50};
int n = sizeof(arr)/sizeof(arr[0]);

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

return 0;
}
```

Output:

```
Array elements: 10 20 30 40 50
```

2. Array Insertion

```c
#include <stdio.h>
int main() {
int arr[6] = {1, 2, 3, 4, 5};
int pos = 2, num = 99, n = 5;

for (int i = n; i > pos; i--)
    arr[i] = arr[i - 1];
arr[pos] = num;
n++;

printf("Array after insertion: ");
```

```c
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
Array after insertion: 1 2 99 3 4 5
```

3. Array Deletion

```c
#include <stdio.h>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int pos = 2, n = 5;

    for (int i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];
    n--;

    printf("Array after deletion: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
Array after deletion: 10 20 40 50
```

Singly Linked List (Insertion, Deletion, Traversal)

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
```

4.

```c
void insert(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
```

5.

```c
void delete(struct Node** head, int key) {
    struct Node *temp = *head, *prev;
    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
```

```c
    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}
void printList(struct Node* head) {
    while (head) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = NULL;
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    printList(head);
    delete(&head, 2);
    printList(head);
    return 0;
}
```

6. Doubly Linked List Implementation

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
void insert(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
```

```
Node));
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;
    if (*head != NULL)
        (*head)->prev = newNode;
    *head = newNode;
}
void printList(struct Node* head) {
    struct Node* last;
    while (head) {
        printf("%d <-> ", head->data);
        last = head;
        head = head->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    printList(head);
    return 0;
}
```

7. Circular Linked List Operations

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
```

```
};
void insert(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    struct Node* temp = *head;
    newNode->data = data;
    newNode->next = *head;
    if (*head != NULL) {
        while (temp->next != *head)
            temp = temp->next;
        temp->next = newNode;
    } else {
        newNode->next = newNode;
    }
    *head = newNode;
}
void printList(struct Node* head) {
    struct Node* temp = head;
    if (head != NULL) {
        do {
            printf("%d -> ", temp->data);
            temp = temp->next;
        } while (temp != head);
    }
    printf("(back to head)\n");
}
int main() {
    struct Node* head = NULL;
    insert(&head, 5);
    insert(&head, 10);
    insert(&head, 15);
    printList(head);
```

```
    return 0;
}
```

8. Stack Implementation using Arrays

```
#include <stdio.h>
#define MAX 100
int stack[MAX], top = -1;
void push(int x) {
    if (top == MAX - 1) printf("Stack Overflow\n");
    else stack[++top] = x;
}
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
void display() {
    if (top == -1) printf("Stack is Empty\n");
    else {
        for (int i = top; i >= 0; i--) printf("%d ", stack[i]);
        printf("\n");
    }
}
int main() {
    push(10); push(20); push(30);
    display(); pop(); display();
    return 0;
}
```

9. Stack Implementation using Linked List

```
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node* next;
} Node;
Node* top = NULL;
void push(int x) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = x; temp->next = top; top = temp;
}
int pop() {
    if (!top) {
        printf("Stack Underflow\n");
        return -1;
    }
    Node* temp = top;
    int val = temp->data;
    top = top->next;
    free(temp);
    return val;
}
void display() {
    Node* temp = top;
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    push(10); push(20); push(30);
```

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

10. Infix to Postfix Conversion

```
#include <ctype.h>
#include <string.h>
#define MAX 100
char stack[MAX];
int top = -1;
void push(char x) { stack[++top] = x; }
char pop() { return (top == -1) ? -1 : stack[top--]; }
int precedence(char x) {
    if (x == '(') return 0;
    if (x == '+' || x == '-') return 1;
    if (x == '*' || x == '/') return 2;
    return -1;
}
void infixToPostfix(char* exp) {
    char *e, x;
    e = exp;
    while (*e != '\0') {
        if (isalnum(*e)) printf("%c", *e);
        else if (*e == '(') push(*e);
        else if (*e == ')') {
            while ((x = pop()) != '(') printf("%c", x);
        }
        else {
            while (precedence(stack[top]) >= precedence(*e))
printf("%c", pop());
            push(*e);
        }
```

```
        e++;
    }
    while (top != -1) printf("%c", pop());
    printf("\n");
}
int main() {
    char exp[MAX];
    printf("Enter infix expression: ");
    scanf("%s", exp);
    infixToPostfix(exp);
    return 0;
}
```

11. Parenthesis Matching using Stack

```
#include <stdbool.h>
bool isBalanced(char* exp) {
    char stack[MAX];
    int top = -1;
    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
stack[++top] = exp[i];
        else {
            if (top == -1) return false;
            char last = stack[top--];
            if ((exp[i] == ')' && last != '(') ||
                (exp[i] == ']' && last != '[') ||
                (exp[i] == '}' && last != '{')) return false;
        }
    }
    return (top == -1);
}
int main() {
```

```c
    char exp[MAX];
    printf("Enter expression: ");
    scanf("%s", exp);
    if (isBalanced(exp)) printf("Balanced\n");
    else printf("Not Balanced\n");
    return 0;
}
```

12. Queue

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

#define MAX 5

// Queue using Arrays
struct Queue {
    int items[MAX];
    int front, rear;
};

void initQueue(struct Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isFull(struct Queue *q) {
    return q->rear == MAX - 1;
}

int isEmpty(struct Queue *q) {
    return q->front == -1 || q->front > q->rear;
}
```

```c
void enqueue(struct Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is Full\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    q->items[++q->rear] = value;
    printf("Inserted %d\n", value);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Deleted %d\n", q->items[q->front++]);
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    for (int i = q->front; i <= q->rear; i++)
        printf("%d ", q->items[i]);
    printf("\n");
}
```

13. Queue using Linked List

```c
struct Node {
    int data;
```

```
    struct Node *next;
};

struct Node *front = NULL, *rear = NULL;

void enqueueLL(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct
Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) front = rear = newNode;
    else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("Inserted %d\n", value);
}

void dequeueLL() {
    if (front == NULL) {
        printf("Queue is Empty\n");
        return;
    }
    struct Node *temp = front;
    printf("Deleted %d\n", temp->data);
    front = front->next;
    free(temp);
    if (front == NULL) rear = NULL;
}

void displayLL() {
    struct Node *temp = front;
```

```c
    if (temp == NULL) {
        printf("Queue is Empty\n");
        return;
    }
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

14.Circular Queue

```c
struct CircularQueue {
    int items[MAX];
    int front, rear;
};

void initCircularQueue(struct CircularQueue *cq) {
    cq->front = cq->rear = -1;
}

int isFullCQ(struct CircularQueue *cq) {
    return (cq->rear + 1) % MAX == cq->front;
}

int isEmptyCQ(struct CircularQueue *cq) {
    return cq->front == -1;
}

void enqueueCQ(struct CircularQueue *cq, int value) {
    if (isFullCQ(cq)) {
        printf("Circular Queue is Full\n");
```

```c
        return;
    }
    if (cq->front == -1) cq->front = 0;
    cq->rear = (cq->rear + 1) % MAX;
    cq->items[cq->rear] = value;
    printf("Inserted %d\n", value);
}

void dequeueCQ(struct CircularQueue *cq) {
    if (isEmptyCQ(cq)) {
        printf("Circular Queue is Empty\n");
        return;
    }
    printf("Deleted %d\n", cq->items[cq->front]);
    if (cq->front == cq->rear) cq->front = cq->rear = -1;
    else cq->front = (cq->front + 1) % MAX;
}

void displayCQ(struct CircularQueue *cq) {
    if (isEmptyCQ(cq)) {
        printf("Circular Queue is Empty\n");
        return;
    }
    int i = cq->front;
    while (1) {
        printf("%d ", cq->items[i]);
        if (i == cq->rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}
```

15. Priority Queue

```
struct PriorityQueue {
    int data[MAX];
    int priority[MAX];
    int size;
};

void initPriorityQueue(struct PriorityQueue *pq) {
    pq->size = 0;
}

void enqueuePQ(struct PriorityQueue *pq, int value, int priority)
{
    if (pq->size == MAX) {
        printf("Priority Queue is Full\n");
        return;
    }
    int i = pq->size++;
    while (i > 0 && pq->priority[i - 1] > priority) {
        pq->data[i] = pq->data[i - 1];
        pq->priority[i] = pq->priority[i - 1];
        i--;
    }
    pq->data[i] = value;
    pq->priority[i] = priority;
    printf("Inserted %d with priority %d\n", value, priority);
}

void dequeuePQ(struct PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is Empty\n");
        return;
```

```
    }
    printf("Deleted %d with priority %d\n", pq->data[0],
pq->priority[0]);
    for (int i = 0; i < pq->size - 1; i++) {
        pq->data[i] = pq->data[i + 1];
        pq->priority[i] = pq->priority[i + 1];
    }
    pq->size--;
}

void displayPQ(struct PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is Empty\n");
        return;
    }
    for (int i = 0; i < pq->size; i++)
        printf("%d(priority %d) ", pq->data[i], pq->priority[i]);
    printf("\n");
}

int main() {
    struct Queue q;
    initQueue(&q);
    enqueue(&q, 10);
    dequeue(&q);
    display(&q);

    enqueueLL(20);
    dequeueLL();
    displayLL();

    struct CircularQueue cq;
```

```
    initCircularQueue(&cq);
    enqueueCQ(&cq, 30);
    dequeueCQ(&cq);
    displayCQ(&cq);

    struct PriorityQueue pq;
    initPriorityQueue(&pq);
    enqueuePQ(&pq, 40, 1);
    dequeuePQ(&pq);
    displayPQ(&pq);
    return 0;
}
```

16. Binary Tree Creation

17. Binary Tree Traversals (Inorder, Preorder, Postorder)

18. Binary Search Tree (BST) Insertion & Search

19. Binary Tree Deletion

20. Height of a Binary Tree

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

// Structure for a node in the binary tree
struct Node {
    int data;
    struct Node *left, *right;
};
```

```c
// Function to create a 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;
}

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

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

    return root;
}

// Function to search a node 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);

    return search(root->right, key);
}
```

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

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

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

// Function to find the minimum value node in a BST
struct Node* findMin(struct Node* root) {
    while (root->left != NULL)
        root = root->left;
```

```
    return root;
}

// Function to delete a node in BST
struct Node* deleteNode(struct Node* root, int key) {
    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) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get the inorder successor
(smallest in the right subtree)
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
```

```
}

// Function to calculate the height of a tree
int height(struct Node* root) {
    if (root == NULL)
        return 0;

    int leftHeight = height(root->left);
    int rightHeight = height(root->right);

    return (leftHeight > rightHeight ? leftHeight : rightHeight)
+ 1;
}

// Main function
int main() {
    struct Node* root = NULL;
    int choice, value, key;

    while (1) {
        printf("\nBinary Tree Operations\n");
        printf("1. Insert\n");
        printf("2. Search\n");
        printf("3. Inorder Traversal\n");
        printf("4. Preorder Traversal\n");
        printf("5. Postorder Traversal\n");
        printf("6. Delete\n");
        printf("7. Height of Tree\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;
            case 2:
                printf("Enter value to search: ");
                scanf("%d", &key);
                if (search(root, key))
                    printf("Node found.\n");
                else
                    printf("Node not found.\n");
                break;
            case 3:
                printf("Inorder Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 4:
                printf("Preorder Traversal: ");
                preorder(root);
                printf("\n");
                break;
            case 5:
                printf("Postorder Traversal: ");
                postorder(root);
                printf("\n");
                break;
            case 6:
                printf("Enter value to delete: ");
                scanf("%d", &key);
```

```
                root = deleteNode(root, key);
                break;
            case 7:
                printf("Height of tree: %d\n", height(root));
                break;
            case 8:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```

21. Graph Representation using Adjacency Matrix

c

CopyEdit

```
#include <stdio.h>

#define V 5 // Number of vertices

void printGraph(int graph[V][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++)
            printf("%d ", graph[i][j]);
        printf("\n");
    }
}

int main() {
```

```
    int graph[V][V] = { {0, 1, 0, 0, 1},
                         {1, 0, 1, 1, 0},
                         {0, 1, 0, 1, 1},
                         {0, 1, 1, 0, 1},
                         {1, 0, 1, 1, 0} };

    printf("Adjacency Matrix Representation of Graph:\n");
    printGraph(graph);
    return 0;
}
```

---

22. Graph Representation using Adjacency List

c

CopyEdit

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

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

struct Graph {
    int numVertices;
    struct Node** adjLists;
};

struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
```

```
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct Node*));

    for (int i = 0; i < vertices; i++)
        graph->adjLists[i] = NULL;

    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\nAdjacency list of vertex %d\n head", v);
        while (temp) {
            printf(" -> %d", temp->vertex);
            temp = temp->next;
```

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

int main() {
    int vertices = 5;
    struct Graph* graph = createGraph(vertices);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printGraph(graph);

    return 0;
}
```

---

23. Breadth-First Search (BFS) Algorithm

c

CopyEdit

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

#define V 6
```

```
struct Queue {
    int items[V];
    int front, rear;
};

struct Queue* createQueue() {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct
Queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

void enqueue(struct Queue* q, int value) {
    if (q->rear == V - 1)
        return;
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

int dequeue(struct Queue* q) {
    int item;
    if (q->front == -1)
        return -1;
    else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
```

```
            q->front = q->rear = -1;
        return item;
    }
}

int isEmpty(struct Queue* q) {
    return q->front == -1;
}

void bfs(int graph[V][V], int startVertex) {
    struct Queue* q = createQueue();
    int visited[V] = {0};

    visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q)) {
        int currentVertex = dequeue(q);
        printf("%d ", currentVertex);

        for (int i = 0; i < V; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(q, i);
            }
        }
    }
}

int main() {
    int graph[V][V] = {
        {0, 1, 1, 0, 0, 0},
```

```
        {1, 0, 1, 1, 0, 0},
        {1, 1, 0, 1, 1, 0},
        {0, 1, 1, 0, 1, 1},
        {0, 0, 1, 1, 0, 1},
        {0, 0, 0, 1, 1, 0}
    };

    printf("BFS Traversal starting from node 0: ");
    bfs(graph, 0);
    return 0;
}
```

---

24. Depth-First Search (DFS) Algorithm

c

CopyEdit

```
#include <stdio.h>

#define V 6

void dfs(int graph[V][V], int vertex, int visited[]) {
    printf("%d ", vertex);
    visited[vertex] = 1;

    for (int i = 0; i < V; i++) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            dfs(graph, i, visited);
        }
    }
}
```

```c
int main() {
    int graph[V][V] = {
        {0, 1, 1, 0, 0, 0},
        {1, 0, 1, 1, 0, 0},
        {1, 1, 0, 1, 1, 0},
        {0, 1, 1, 0, 1, 1},
        {0, 0, 1, 1, 0, 1},
        {0, 0, 0, 1, 1, 0}
    };

    int visited[V] = {0};
    printf("DFS Traversal starting from node 0: ");
    dfs(graph, 0, visited);

    return 0;
}
```

25. Trie Implementation (Insert, Search, Delete)

c

CopyEdit

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

#define ALPHABET_SIZE 26

// Trie node structure
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];
    int isEndOfWord;
};
```

```c
// Create a new Trie node
struct TrieNode* getNode() {
    struct TrieNode* node = (struct
TrieNode*)malloc(sizeof(struct TrieNode));
    node->isEndOfWord = 0;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        node->children[i] = NULL;
    }
    return node;
}

// Insert a word into the Trie
void insert(struct TrieNode* root, const char* word) {
    struct TrieNode* node = root;
    while (*word) {
        int index = *word - 'a';
        if (!node->children[index]) {
            node->children[index] = getNode();
        }
        node = node->children[index];
        word++;
    }
    node->isEndOfWord = 1;
}

// Search for a word in the Trie
int search(struct TrieNode* root, const char* word) {
    struct TrieNode* node = root;
    while (*word) {
        int index = *word - 'a';
        if (!node->children[index]) {
```

```
            return 0;  // Not found
        }
        node = node->children[index];
        word++;
    }
    return node != NULL && node->isEndOfWord;
}

int main() {
    struct TrieNode* root = getNode();
    insert(root, "hello");
    insert(root, "hell");
    printf("Search for 'hello': %d\n", search(root, "hello"));
    printf("Search for 'hell': %d\n", search(root, "hell"));
    return 0;
}
```

26. String Matching using KMP Algorithm

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

void computeLPSArray(char* pattern, int M, int* lps) {
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < M) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
```

```
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

void KMPSearch(char* text, char* pattern) {
    int M = strlen(pattern);
    int N = strlen(text);
    int lps[M];
    computeLPSArray(pattern, M, lps);

    int i = 0;  // index for text
    int j = 0;  // index for pattern
    while (i < N) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }
        if (j == M) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        } else if (i < N && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
```

```
                i++;
            }
        }
    }
}

int main() {
    char text[] = "ABABDABACDABABCABAB";
    char pattern[] = "ABABCABAB";
    KMPSearch(text, pattern);
    return 0;
}
```

27. Longest Common Subsequence (LCS)

c

CopyEdit

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

int max(int a, int b) {
    return a > b ? a : b;
}

int LCS(char* X, char* Y, int m, int n) {
    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
```

```
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp[m][n];
}

int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    int m = strlen(X);
    int n = strlen(Y);
    printf("Length of LCS: %d\n", LCS(X, Y, m, n));
    return 0;
}
```

28. Dynamic Programming: Fibonacci Memoization

c

CopyEdit

```
#include <stdio.h>

#define MAX 1000
int memo[MAX];

int fib(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```

```c
int main() {
    for (int i = 0; i < MAX; i++) memo[i] = -1;
    printf("Fibonacci of 10: %d\n", fib(10));
    return 0;
}
```

29. Dynamic Programming: 0/1 Knapsack Problem

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

int max(int a, int b) {
    return a > b ? a : b;
}

int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (wt[i - 1] <= w) {
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i -
1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][W];
}
```

```
int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    printf("Maximum value in Knapsack: %d\n", knapsack(W, wt,
val, n));
    return 0;
}
```

30. Segment Tree for Range Sum Query

c

CopyEdit

```
#include <stdio.h>

void buildSegmentTree(int *arr, int *segTree, int low, int high,
int pos) {
    if (low == high) {
        segTree[pos] = arr[low];
    } else {
        int mid = (low + high) / 2;
        buildSegmentTree(arr, segTree, low, mid, 2 * pos + 1);
        buildSegmentTree(arr, segTree, mid + 1, high, 2 * pos +
2);
        segTree[pos] = segTree[2 * pos + 1] + segTree[2 * pos +
2];
    }
}

int rangeSumQuery(int *segTree, int qlow, int qhigh, int low, int
```

```
high, int pos) {
    if (qlow <= low && qhigh >= high) return segTree[pos];
    if (qlow > high || qhigh < low) return 0;
    int mid = (low + high) / 2;
    return rangeSumQuery(segTree, qlow, qhigh, low, mid, 2 * pos
+ 1) +
           rangeSumQuery(segTree, qlow, qhigh, mid + 1, high, 2 *
pos + 2);
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    int segTree[4 * n];
    buildSegmentTree(arr, segTree, 0, n - 1, 0);
    printf("Range sum (1, 3): %d\n", rangeSumQuery(segTree, 1, 3,
0, n - 1, 0));
    return 0;
}
```

31. Fenwick Tree (Binary Indexed Tree)

c

CopyEdit

```
#include <stdio.h>

void update(int *bit, int n, int index, int value) {
    while (index <= n) {
        bit[index] += value;
        index += index & (-index);
    }
}
```

```
int query(int *bit, int index) {
    int sum = 0;
    while (index > 0) {
        sum += bit[index];
        index -= index & (-index);
    }
    return sum;
}

int main() {
    int arr[] = {1, 3, 4, 8, 6, 1, 4, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    int bit[n + 1];
    for (int i = 1; i <= n; i++) bit[i] = 0;

    for (int i = 0; i < n; i++) update(bit, n, i + 1, arr[i]);

    printf("Prefix sum of first 3 elements: %d\n", query(bit,
3));
    return 0;
}
```