DATA STRUCTURES IN C

**WHAT IS DATA STRUCTURE?**

**Data structures** are essential components that help organize and store data efficiently in computer memory. They provide a way to manage and manipulate data effectively, enabling faster access, insertion, and deletion operations.

1. Arrays

An array is a collection of elements of the same type placed in contiguous memory locations.

 Operations

1)Accessing Elements: Direct access using index.

2)Insertion: Adding an element at a specific index (shifts elements).

3)Deletion: Removing an element from a specific index (shifts elements).

4)Search: Finding an element (linear or binary search).

 Sample Code

.c

#include <stdio.h>

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    // Accessing elements

    printf("Element at index 2: %d\n", arr[2]);

    // Inserting element at index 2

    for(int i = 4; i >= 2; i--) {

        arr[i + 1] = arr[i];

    }

    arr[2] = 99;

    printf("After insertion: ");

    for(int i = 0; i < 6; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

    // Deleting element at index 2

    for(int i = 2; i < 5; i++) {

        arr[i] = arr[i + 1];

    }

    printf("After deletion: ");

    for(int i = 0; i < 5; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

    return 0;

}

 Use Case

- Storing fixed number of elements, like monthly temperatures.

 Time Complexity

- Access: O(1)

- Insertion: O(n)

- Deletion: O(n)

- Search: O(n) (linear search), O(log n) (binary search)

2. Linked Lists

A linked list is a sequence of nodes where each node contains data and a reference to the next node.

 Operations

1)Traversal: Accessing each element sequentially.

2)Insertion: Adding a new node.

3)Deletion: Removing an existing node.

4)Search: Finding an element.

 Sample Code

.c

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

};

void printList(struct Node\* n) {

    while (n != NULL) {

        printf("%d ", n->data);

        n = n->next;

    }

    printf("\n");

}

void insertAtBeginning(struct Node\*\* head, int new\_data) {

    struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

    new\_node->data = new\_data;

    new\_node->next = (\*head);

    (\*head) = new\_node;

}

void deleteNode(struct Node\*\* head, int key) {

    struct Node\* temp = \*head;

    struct Node\* prev = NULL;

    if (temp != NULL && temp->data == key) {

        \*head = temp->next;

        free(temp);

        return;

    }

    while (temp != NULL && temp->data != key) {

        prev = temp;

        temp = temp->next;

    }

    if (temp == NULL) return;

    prev->next = temp->next;

    free(temp);

}

int main() {

    struct Node\* head = NULL;

    insertAtBeginning(&head, 1);

    insertAtBeginning(&head, 2);

    insertAtBeginning(&head, 3);

    printf("Linked list: ");

    printList(head);

    deleteNode(&head, 2);

    printf("After deletion: ");

    printList(head);

    return 0;

}

Use Case

- Dynamic memory allocation, such as in a playlist of songs.

 Time Complexity

- Access: O(n)

- Insertion: O(1) (at beginning)

- Deletion: O(n)

- Search: O(n)

3. Stacks

A stack is a collection of elements that follows the Last In First Out (LIFO) principle.

 Operations

1)Push: Add an element to the top.

2)Pop: Remove the top element.

3)Peek: Access the top element.

Sample Code

.c

#include <stdio.h>

#include <stdlib.h>

#define MAX 1000

struct Stack {

    int top;

    int arr[MAX];

};

struct Stack\* createStack() {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->top = -1;

    return stack;

}

void push(struct Stack\* stack, int item) {

    if (stack->top == MAX - 1) {

        printf("Stack overflow\n");

        return;

    }

    stack->arr[++stack->top] = item;

}

int pop(struct Stack\* stack) {

    if (stack->top == -1) {

        printf("Stack underflow\n");

        return -1;

    }

    return stack->arr[stack->top--];

}

int peek(struct Stack\* stack) {

    if (stack->top == -1) {

        printf("Stack is empty\n");

        return -1;

    }

    return stack->arr[stack->top];

}

int main() {

    struct Stack\* stack = createStack();

    push(stack, 10);

    push(stack, 20);

    push(stack, 30);

    printf("Top element: %d\n", peek(stack));

    printf("Elements: %d %d %d\n", pop(stack), pop(stack), pop(stack));

    return 0;

}

Use Case

- Function call management (call stack in programming languages).

Time Complexity

- Push: O(1)

- Pop: O(1)

- Peek: O(1)

4. Queues

Description

A queue is a collection of elements that follows the First In First Out (FIFO) principle.

Operations

1)Enqueue: Add an element to the end.

2)Dequeue: Remove the front element.

3)Peek: Access the front element.

 Sample Code

.c

#include <stdio.h>

#include <stdlib.h>

struct Queue {

    int front, rear, size;

    unsigned capacity;

    int\* array;

};

struct Queue\* createQueue(unsigned capacity) {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->capacity = capacity;

    queue->front = queue->size = 0;

    queue->rear = capacity - 1;

    queue->array = (int\*)malloc(queue->capacity \* sizeof(int));

    return queue;

}

int isFull(struct Queue\* queue) {

    return (queue->size == queue->capacity);

}

int isEmpty(struct Queue\* queue) {

    return (queue->size == 0);

}

void enqueue(struct Queue\* queue, int item) {

    if (isFull(queue)) return;

    queue->rear = (queue->rear + 1) % queue->capacity;

    queue->array[queue->rear] = item;

    queue->size = queue->size + 1;

}

int dequeue(struct Queue\* queue) {

    if (isEmpty(queue)) return -1;

    int item = queue->array[queue->front];

    queue->front = (queue->front + 1) % queue->capacity;

    queue->size = queue->size - 1;

    return item;

}

int front(struct Queue\* queue) {

    if (isEmpty(queue)) return -1;

    return queue->array[queue->front];

}

int rear(struct Queue\* queue) {

    if (isFull(queue)) return -1;

    return queue->array[queue->rear];

}

int main() {

    struct Queue\* queue = createQueue(1000);

    enqueue(queue, 10);

    enqueue(queue, 20);

    enqueue(queue, 30);

    printf("Front element: %d\n", front(queue));

    printf("Dequeued element: %d\n", dequeue(queue));

    return 0;

}

 Use Case

- Print queue management in operating systems.

 Time Complexity

- Enqueue: O(1)

- Dequeue: O(1)

- Peek: O(1)

5. Trees (Binary Search Tree)

 Description

A tree is a hierarchical data structure with nodes. A binary search tree (BST) is a tree where each node has at most two children, and the left child is less than the parent, and the right child is greater.

Operations

1)Insertion: Adding a new node.

2)Deletion: Removing a node.

3)Search: Finding a node.

4)Traversal: In-order, Pre-order, Post-order.

 Sample Code

.c

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

struct Node\* newNode(int data) {

    struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

    node->data = data;

    node->left = node->right = NULL;

    return node;

}

struct Node\* insert(struct Node\* node, int data) {

    if (node == NULL) return newNode(data);

    if (data < node->data)

        node->left = insert(node->left, data);

    else if (data > node->data)

        node->right = insert(node->right, data);

    return node;

 inOrder(root->left);

        printf(“%d “, root->data);

        inOrder(root->right);

    }

}

Struct Node\* minValueNode(struct Node\* node) {

    Struct Node\* current = node;

    While (current && current->left != NULL)

        Current = current->left;

    Return current;

}

Struct Node\* deleteNode(struct Node\* root, int data) {

    If (root == NULL) return root;

    If (data < root->data)

        Root->left = deleteNode(root->left, data);

    Else if (data > root->data)

        Root->right = deleteNode(root->right, data);

    Else {

        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;

        }

        Struct Node\* temp = minValueNode(root->right);

        Root->data = temp->data;

        Root->right = deleteNode(root->right, temp->data);

    }

    Return root;

}

Int main() {

    Struct Node\* root = NULL;

    Root = insert(root, 50);

    Insert(root, 30);

    Insert(root, 20);

    Insert(root, 40);

    Insert(root, 70);

    Insert(root, 60);

    Insert(root, 80);

    Printf(“In-order traversal: “);

    inOrder(root);

    printf(“\n”);

    printf(“Deleting 20\n”);

    root = deleteNode(root, 20);

    printf(“In-order traversal: “);

    inOrder(root);

    printf(“\n”);

    return 0;

}

 Use Case

-Hierarchical data storage, such as file systems.

Time Complexity

- Insertion: O(h)

- Deletion: O(h)

- Search: O(h)

- Traversal: O(n)

  \*h is the height of the tree