**DATA STRUCTURES IN C**

**1.** **Arrays**

**Definition-**

An array is a data structure that stores a collection of elements, typically of the same data type, in a contiguous block of memory. Each element in the array can be accessed using an index or a key. Arrays are commonly used in programming because they allow for efficient storage and retrieval of data

Key characteristics of arrays include:

Fixed Size: The size of an array is defined at the time of its creation and cannot be changed dynamically.

Indexed Access: Each element in an array is accessed by its index, usually starting from zero.

Homogeneous Elements: All elements in an array are of the same data type (e.g., all integers, all floats, all strings).

.

**Operations-**

* **Access**: O(1)
* **Search**: O(n)
* **Insert**: O(n) (worst case, shifting elements)
* **Delete**: O(n) (worst case, shifting elements)

**Use Case-**

When you need fast access by index.

**Example Code-**

#include <stdio.h>

int main() {

int arr[5] = {1, 2, 3, 4, 5};

// Access element

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

// Insert element (requires shifting)

int n = 5;

for (int i = n; i > 2; i--) {

arr[i] = arr[i - 1];

}

arr[2] = 10;

n++;

// Print array

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

return 0;

}

**2. Linked Lists**

**Definition-**

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

**Types-**

* **Singly Linked List**: A singly linked list is a linear data structure consisting of a sequence of elements, where each element is a separate object called a node. Each node contains two components:
* 1. **\*Data\***: The actual value or information stored in the node.
* 2**. \*Next\***: A reference or pointer to the next node in the sequence.
* **Doubly Linked List** : A doubly linked list is a type of linked list in which each node contains three components:
* 1. **\*Data\***: The actual value or information stored in the node.
* 2. **\*Next\***: A reference or pointer to the next node in the sequence.
* 3**. \*Prev\***: A reference or pointer to the previous node in the sequence.
* **Circular Linked List** : A circular linked list is a variation of the linked list where the last node points back to the first node, forming a circle. This structure can be beneficial in situations where it is desirable to loop through the list repeatedly without using additional pointers or conditions to reset to the beginning.

**Operations-**

* **Access**: O(n)
* **Search**: O(n)
* **Insert**: O(1) (if inserting at head)
* **Delete**: O(1) (if deleting at head)

**Use Case-**

When you need efficient insertions/deletions.

**Example Code-**

#include <stdio.h>

#include <stdlib.h>

// Define the node structure

struct Node {

int data;

struct Node\* next;

};

// Insert at the head

void insertAtHead(struct Node\*\* head, int newData) {

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

newNode->data = newData;

newNode->next = \*head;

\*head = newNode;

}

// Print the linked list

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d -> ", node->data);

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtHead(&head, 1);

insertAtHead(&head, 2);

insertAtHead(&head, 3);

printList(head);

return 0;

}

**3. Stacks**

**Definition-**

A stack is a linear data structure that operates based on the Last In, First Out (LIFO) principle. In a stack, the most recently added element is the first one to be removed. Think of it as a collection of items similar to a stack of plates: you can only add or remove plates from the top of the stack.

**Operations-**

* **Push (insert)**: O(1)
* **Pop (remove)**: O(1)
* **Peek (top element)**: O(1)

**Use Case-**

Managing function calls, recursive algorithms, undo mechanisms.

**Example Code:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX 100 // Maximum size of the stack

// Stack structure definition

typedef struct Stack {

int top;

int array[MAX];

} Stack; // Function to create a stack

Stack\* createStack() {

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

stack->top = -1;

return stack;

}

// Function to check if the stack is full

int isFull(Stack\* stack) {

return stack->top == MAX - 1;

}

// Function to check if the stack is empty

int isEmpty(Stack\* stack) {

return stack->top == -1;

}

// Function to add an item to the stack

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

if (isFull(stack)) {

printf("Stack overflow. Cannot push %d\n", item);

return;

}

stack->array[++stack->top] = item;

printf("%d pushed to stack\n", item);

}

// Function to remove an item from the stack

int pop(Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack underflow. Cannot pop\n");

return INT\_MIN;

}

return stack->array[stack->top--];

}

// Function to get the top item of the stack

int peek(Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty. Cannot peek\n");

return INT\_MIN;

}

return stack->array[stack->top];

}

// Function to display the stack

void display(Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return;

}

printf("Stack: ");

for (int i = 0; i <= stack->top; i++) {

printf("%d ", stack->array[i]);

}

printf("\n");

}

// Main function to demonstrate stack operations

int main() {

Stack\* stack = createStack();

push(stack, 10);

push(stack, 20);

push(stack, 30);

display(stack); // Output: Stack: 10 20 30

printf("Popped from stack: %d\n", pop(stack)); // Output: Popped from stack: 30

display(stack); // Output: Stack: 10 20

printf("Top element is: %d\n", peek(stack)); // Output: Top element is: 20

return 0;

}

**4. Queues**

**Definition-**

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. In a queue, elements are added to the rear (also known as enqueue) and removed from the front (also known as dequeue). Think of it as a line of people waiting for a service: the person who joined the line first gets served first.

**Types-**

* Simple Queue
* Circular Queue
* Priority Queue
* Deque

**Operations-**

* **Enqueue (insert)**: O(1)
* **Dequeue (remove)**: O(1)
* **Peek (front element)**: O(1)

**Use Case-**

Task scheduling, handling requests in web servers, breadth-first search (BFS).

**Example Code-**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum size of the queue

// Queue structure definition

typedef struct Queue {

int front, rear, size;

unsigned capacity;

int\* array;

} Queue;

// Function to create a queue of given capacity

Queue\* createQueue(unsigned capacity) {

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

queue->capacity = capacity;

queue->front = queue->size = 0;

queue->rear = capacity - 1; // This is important, see the enqueue

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

return queue;

}

// Function to check if the queue is full

int isFull(Queue\* queue) {

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

}

// Function to check if the queue is empty

int isEmpty(Queue\* queue) {

return (queue->size == 0);

}

// Function to add an item to the queue

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

if (isFull(queue)) {

printf("Queue overflow. Cannot enqueue %d\n", item);

return;

}

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

queue->array[queue->rear] = item;

queue->size = queue->size + 1;

printf("%d enqueued to queue\n", item);

}

// Function to remove an item from the queue

int dequeue(Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue underflow. Cannot dequeue\n");

return INT\_MIN;

}

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

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

queue->size = queue->size - 1;

return item;

}

// Function to get the front item of the queue

int front(Queue\* queue) {

if (isEmpty(queue)) {

return INT\_MIN;

}

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

}

// Function to get the rear item of the queue

int rear(Queue\* queue) {

if (isFull(queue)) {

return INT\_MIN;

}

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

}

// Function to display the queue

void display(Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty\n");

return;

}

printf("Queue: ");

for (int i = queue->front; i != queue->rear; i = (i + 1) % queue->capacity) {

printf("%d ", queue->array[i]);

}

printf("%d\n", queue->array[queue->rear]);

}

// Main function to demonstrate queue operations

int main() {

Queue\* queue = createQueue(MAX);

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

enqueue(queue, 40);

display(queue); // Output: Queue: 10 20 30 40

printf("Dequeued from queue: %d\n", dequeue(queue)); // Output: Dequeued from queue: 10

display(queue); // Output: Queue: 20 30 40

printf("Front item is: %d\n", front(queue)); // Output: Front item is: 20

printf("Rear item is: %d\n", rear(queue)); // Output: Rear item is: 40

return 0;

}

**5. Trees**

**Definition-**

A tree is a hierarchical structure with nodes, with one node as the root and zero or more child nodes.

**Types-**

* Binary Tree
* Binary Search Tree (BST)
* AVL Tree
* Red-Black Tree
* B-trees

**Operations (BST)-**

* **Access**: O(log n)
* **Search**: O(log n)
* **Insert**: O(log n)
* **Delete**: O(log n)

**Use Case-**

Hierarchical data representation, efficient data retrieval, database indexing.

**Example Code for BST-**

#include <stdio.h>

#include <stdlib.h>

// Define the node structure

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Create a new node

struct Node\* newNode(int data) {

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

node->data = data;

node->left = node->right = NULL;

return node;

}

// Insert a new node in BST

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 traversal of BST

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

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);

inorder(root);

return 0;

}

**6. Heaps**

**Definition-**

A heap is a specialized binary tree-based data structure that satisfies the heap property. In a heap, the value of a parent node is either greater than or equal to (max heap) or less than or equal to (min heap) the values of its children. This property ensures that the root node is either the maximum or minimum element in the heap

**Types-**

* Min-Heap
* Max-Heap

**Operations-**

* **Insert**: O(log n)
* **Delete (root)**: O(log n)
* **Peek (min/max)**: O(1)

**Use Case-**

Implementing priority queues, scheduling algorithms, heap sort.

**Example Code for Heap-**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Function to swap two elements

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted with the given node

void maxHeapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected sub-tree

maxHeapify(arr, n, largest);

}

}

// Function to build a max heap from an array

void buildMaxHeap(int arr[], int n) {

// Start from the last non-leaf node

for (int i = (n / 2) - 1; i >= 0; i--) {

maxHeapify(arr, n, i);

}

}

// Function to insert a new element into the heap

void insert(int arr[], int\* n, int key) {

if (\*n == MAX\_SIZE) {

printf("Heap overflow. Cannot insert %d\n", key);

return;

}

// Insert the new element at the end

(\*n)++;

int i = \*n - 1;

arr[i] = key;

// Fix the max heap property if violated

while (i != 0 && arr[(i - 1) / 2] < arr[i]) {

swap(&arr[i], &arr[(i - 1) / 2]);

i = (i - 1) / 2;

}

printf("%d inserted into heap\n", key);

}

// Function to extract the maximum element from the heap

int extractMax(int arr[], int\* n) {

if (\*n <= 0) {

printf("Heap underflow. Cannot extract maximum\n");

return -1;

}

if (\*n == 1) {

(\*n)--;

return arr[0];

}

// Store the maximum value and remove it from the heap

int max = arr[0];

arr[0] = arr[\*n - 1];

(\*n)--;

// Heapify the root

maxHeapify(arr, \*n, 0);

return max;

}

// Main function to demonstrate heap operations

int main() {

int arr[MAX\_SIZE] = {10, 20, 15, 25, 30};

int n = 5;

printf("Original heap: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

buildMaxHeap(arr, n);

printf("Max heap after build: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

insert(arr, &n, 50);

printf("Max heap after insertion: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

printf("Extracted maximum element: %d\n", extractMax(arr, &n));

printf("Max heap after extraction: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**7. Hash Tables**

**Definition-**

A hash table is a collection of key-value pairs with efficient key-based access.

**Operations-**

* **Access**: O(1) (average case)
* **Search**: O(1) (average case)
* **Insert**: O(1) (average case)
* **Delete**: O(1) (average case)

**Use Case-**

Fast lookup, insert, and delete operations, like dictionaries, caches.

**Example Code-**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 10

struct HashNode {

int key;

int value;

struct HashNode\* next;

};

struct HashTable {

struct HashNode\* table[TABLE\_SIZE];

};

// Hash function

int hashFunction(int key) {

return key % TABLE\_SIZE;

}

// Insert key-value pair

void insert(struct HashTable\* ht, int key, int value) {

int hashIndex = hashFunction(key);

struct HashNode\* newNode = (struct HashNode\*)malloc(sizeof(struct HashNode));

newNode->key = key;

newNode->value = value;

newNode->next = NULL;

if (ht->table[hashIndex] == NULL) {

ht->table[hashIndex] = newNode;

} else {

struct HashNode\* temp = ht->table[hashIndex];

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Search for a key

int search(struct HashTable\* ht, int key) {

int hashIndex = hashFunction(key);

struct HashNode\* temp = ht->table[hashIndex];

while (temp != NULL) {

if (temp->key == key) {

return temp->value;

}

temp = temp->next;

}

return -1;

}

// Delete a key

void delete(struct HashTable\* ht, int key) {

int hashIndex = hashFunction(key);

struct HashNode\* temp = ht->table[hashIndex];

struct HashNode\* prev = NULL;

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

prev = temp;

temp = temp->next;

}

if (temp == NULL) {

printf("Key not found\n");

return;

}

if (prev == NULL) {

ht->table[hashIndex] = temp->next;

} else {

prev->next = temp->next;

}

free(temp);

}

int main() {

struct HashTable\* ht = (struct HashTable\*)malloc(sizeof(struct HashTable));

memset(ht->table, 0, sizeof(ht->table));

insert(ht, 1, 10);

insert(ht, 2, 20);

insert(ht, 3, 30);

printf("Value for key 2: %d\n", search(ht, 2));

delete(ht, 2);

printf("Value for key 2 after deletion: %d\n", search(ht, 2));

return 0;

}

**8. Graphs**

**Definition-**

A graph is a collection of nodes (vertices) and edges connecting pairs of nodes.

**Types-**

* Directed
* Undirected
* Weighted
* Unweighted

**Representations-**

* Adjacency List
* Adjacency Matrix

**Operations-**

* **Add Vertex**: O(1)
* **Add Edge**: O(1) (adjacency list), O(1) (adjacency matrix)
* **Remove Vertex**: O(V + E) (adjacency list), O(V^2) (adjacency matrix)
* **Remove Edge**: O(E) (adjacency list), O(1) (adjacency matrix)

**Use Case-**

Modeling networks like social media, transportation systems, or computer networks.

**Example Code for GRAPHS:**

#include <stdio.h>

#include <stdlib.h>

// Structure for adjacency list node

typedef struct AdjListNode {

int dest;

struct AdjListNode\* next;

} AdjListNode;

// Structure for adjacency list

typedef struct AdjList {

AdjListNode\* head;

} AdjList;

// Structure for graph with array of adjacency lists

typedef struct Graph {

int V; // Number of vertices

AdjList\* array;

} Graph;

// Function to create a new adjacency list node

AdjListNode\* newAdjListNode(int dest) {

AdjListNode\* newNode = (AdjListNode\*)malloc(sizeof(AdjListNode));

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with V vertices

Graph\* createGraph(int V) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->V = V;

// Create an array of adjacency lists, size of V

graph->array = (AdjList\*)malloc(V \* sizeof(AdjList));

// Initialize each adjacency list as empty by making head as NULL

for (int i = 0; i < V; ++i) {

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(Graph\* graph, int src, int dest) {

// Add an edge from src to dest. A new node is added to the adjacency list of src.

AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Since graph is undirected, add an edge from dest to src also

newNode = newAdjListNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to print the adjacency list representation of graph

void printGraph(Graph\* graph) {

for (int v = 0; v < graph->V; ++v) {

AdjListNode\* pCrawl = graph->array[v].head;

printf("\n Adjacency list of vertex %d\n head ", v);

while (pCrawl) {

printf("-> %d", pCrawl->dest);

pCrawl = pCrawl->next;

}

printf("\n");

}

}

// Function to perform Depth First Search traversal of the graph

void DFSUtil(int v, Graph\* graph, int visited[]) {

visited[v] = 1;

printf("%d ", v);

AdjListNode\* pCrawl = graph->array[v].head;

while (pCrawl) {

if (!visited[pCrawl->dest]) {

DFSUtil(pCrawl->dest, graph, visited);

}

pCrawl = pCrawl->next;

}

}

// Function to initiate Depth First Search traversal of the graph

void DFS(Graph\* graph) {

int\* visited = (int\*)malloc(graph->V \* sizeof(int));

for (int i = 0; i < graph->V; i++) {

visited[i] = 0;

}

printf("Depth First Search Traversal of the graph: ");

for (int i = 0; i < graph->V; i++) {

if (!visited[i]) {

DFSUtil(i, graph, visited);

}

}

printf("\n");

}

// Main function

int main() {

// Create a graph with 5 vertices

int V = 5;

Graph\* graph = createGraph(V);

// Add edges

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

// Print the adjacency list representation of the graph

printf("Adjacency List Representation of the graph:\n");

printGraph(graph);

// Perform Depth First Search traversal of the graph

DFS(graph);

return 0;

}

**Common Algorithmic Patterns-**

**Traversal-**

* **Arrays**: For loops, iterators
* **Linked Lists**: While loop until null
* **Trees**: Depth-First Search (DFS) (Preorder, Inorder, Postorder), Breadth-First Search (BFS)
* **Graphs**: DFS, BFS

**Sorting-**

* **Arrays**: QuickSort, MergeSort, BubbleSort, etc.
* **Linked Lists**: MergeSort (efficient for linked lists)

**Searching-**

* **Arrays**: Binary Search (sorted arrays)
* **Trees**: Binary Search Tree search
* **Graphs**: DFS, BFS

**Time Complexity Cheats-**

* **Accessing Array Element**: O(1)
* **Searching in Unsorted Array**: O(n)
* **Binary Search**: O(log n)
* **Linked List Insertion/Deletion**: O(1) (if pointer to node is known)
* **BST Operations (Search, Insert, Delete)**: O(log n) average, O(n) worst-case
* **Heap Operations (Insert, Delete)**: O(log n)
* **Hash Table Operations (Search, Insert, Delete)**: O(1) average, O(n) worst-case