**Data Structures in C**

**1. Arrays**

**Definition-**

An array is a collection of elements identified by index or key.

**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
* Doubly Linked List
* Circular Linked List

**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 collection of elements with Last In, First Out (LIFO) access.

**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>

// Define the stack structure struct StackNode {

int data;

struct StackNode\* next;

};

// Push operation

void push(struct StackNode\*\* top, int newData) {

struct StackNode\* newNode = (struct StackNode\*)malloc(sizeof(struct StackNode)); newNode->data = newData;

newNode->next = \*top;

\*top = newNode;

}

// Pop operation

int pop(struct StackNode\*\* top) { if (\*top == NULL) {

printf("Stack underflow\n"); return -1;

}

int popped = (\*top)->data; struct StackNode\* temp = \*top;

\*top = (\*top)->next; free(temp);

return popped;

}

// Peek operation

int peek(struct StackNode\* top) { if (top == NULL) {

return -1;

}

return top->data;

}

int main() {

struct StackNode\* stack = NULL; push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

printf("Top element is %d\n", peek(stack)); printf("Popped element is %d\n", pop(&stack)); printf("Top element is %d\n", peek(stack)); return 0;

}

# 4. Queues

## Definition-

A queue is a collection of elements with First In, First Out (FIFO) access.

## 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 the queue node structure struct QueueNode {

int data;

struct QueueNode\* next;

};

// Define the queue structure struct Queue {

struct QueueNode \*front, \*rear;

};

// Create a new queue node

struct QueueNode\* newNode(int k) {

struct QueueNode\* temp = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

temp->data = k; temp->next = NULL; return temp;

}

// Create an empty queue struct Queue\* createQueue() {

struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue)); q->front = q->rear = NULL;

return q;

}

// Enqueue operation

void enqueue(struct Queue\* q, int k) { struct QueueNode\* temp = newNode(k); if (q->rear == NULL) {

q->front = q->rear = temp; return;

}

q->rear->next = temp; q->rear = temp;

}

// Dequeue operation

int dequeue(struct Queue\* q) { if (q->front == NULL) {

return -1;

}

struct QueueNode\* temp = q->front; int data = temp->data;

q->front = q->front->next; if (q->front == NULL) {

q->rear = NULL;

}

free(temp); return data;

}

int main() {

struct Queue\* q = createQueue(); enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

printf("Dequeued element is %d\n", dequeue(q)); printf("Dequeued element is %d\n", dequeue(q)); 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 special tree-based structure that satisfies the heap property.

**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 Min-Heap-**

#define MAX\_HEAP\_SIZE 100

struct MinHeap { int size;

int array[MAX\_HEAP\_SIZE];

};

// Function to swap two elements void swap(int\* x, int\* y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to heapify the node at index i

void minHeapify(struct MinHeap\* minHeap, int i) { int smallest = i;

int left = 2 \* i + 1; int right = 2 \* i + 2;

if (left < minHeap->size && minHeap->array[left] < minHeap->array[smallest]) smallest = left;

if (right < minHeap->size && minHeap->array[right] < minHeap->array[smallest]) smallest = right;

if (smallest != i) {

swap(&minHeap->array[i], &minHeap->array[smallest]); minHeapify(minHeap, smallest);

}

}

// Function to insert a new key 'key'

void insertKey(struct MinHeap\* minHeap, int key) { if (minHeap->size == MAX\_HEAP\_SIZE) {

printf("Overflow: Could not insert key\n"); return;

}

minHeap->size++;

int i = minHeap->size - 1; minHeap->array[i] = key;

// Fix the min heap property if it is violated

while (i != 0 && minHeap->array[i] < minHeap->array[(i - 1) / 2]) { swap(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

// Function to extract the root which is the minimum element int extractMin(struct MinHeap\* minHeap) {

if (minHeap->size <= 0) return INT\_MAX;

if (minHeap->size == 1) { minHeap->size--;

return minHeap->array[0];

}

int root = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1]; minHeap->size--;

minHeapify(minHeap, 0);

return root;

}

// Function to create a Min Heap struct MinHeap\* createMinHeap() {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap)); minHeap->size = 0;

return minHeap;

}

int main() {

struct MinHeap\* minHeap = createMinHeap(); insertKey(minHeap, 3);

insertKey(minHeap, 2);

insertKey(minHeap, 1);

insertKey(minHeap, 15);

insertKey(minHeap, 5);

insertKey(minHeap, 4);

insertKey(minHeap, 45);

printf("Extracted min value: %d\n", extractMin(minHeap)); printf("Extracted min value: %d\n", extractMin(minHeap)); printf("Extracted min value: %d\n", extractMin(minHeap));

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;

}

# 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 Adjacency List Representation-**

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

// Define the structure for an adjacency list node struct AdjListNode {

int dest;

struct AdjListNode\* next;

};

// Define the structure for an adjacency list struct AdjList {

struct AdjListNode\* head;

};

// Define the structure for a graph struct Graph {

int V;

struct AdjList\* array;

};

// Create a new adjacency list node

struct AdjListNode\* newAdjListNode(int dest) {

struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode)); newNode->dest = dest;

newNode->next = NULL; return newNode;

}

// Create a graph of V vertices struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph)); graph->V = V;

graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList)); for (int i = 0; i < V; ++i) {

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

}

return graph;

}

// Add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) { struct AdjListNode\* newNode = newAdjListNode(dest); newNode->next = graph->array[src].head;

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

newNode = newAdjListNode(src);

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

}

// Print the adjacency list representation of the graph void printGraph(struct Graph\* graph) {

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

struct 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");

}

}

int main() { int V = 5;

struct Graph\* graph = createGraph(V); 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;

}

# Tries (Prefix Trees)

## Definition-

A trie is a tree-like data structure that stores a dynamic set of strings, typically used for searching.

**Operations-**

|  |  |  |
| --- | --- | --- |
|  | **Insert**: | O(m) where m is the length of the word  **h**: O(m)  : O(m) |
| **Searc** |
| **Delete** |

## Use Case-

Efficiently searching and storing a large set of strings, autocomplete systems.

## Example Code-

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

#define ALPHABET\_SIZE 26

// Define the structure for a trie node struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE]; int isEndOfWord;

};

// Create a new trie node struct TrieNode\* getNode() {

struct TrieNode\* pNode = (struct TrieNode\*)malloc(sizeof(struct TrieNode)); pNode->isEndOfWord = 0;

for (int i = 0; i < ALPHABET\_SIZE; i++) pNode->children[i] = NULL;

return pNode;

}

// Insert a word into the trie

void insert(struct TrieNode\* root, const char\* key) { struct TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) { int index = key[level] - 'a';

if (!pCrawl->children[index])

pCrawl->children[index] = getNode(); pCrawl = pCrawl->children[index];

}

pCrawl->isEndOfWord = 1;

}

// Search for a word in the trie

int search(struct TrieNode\* root, const char\* key) { struct TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) { int index = key[level] - 'a';

if (!pCrawl->children[index]) return 0;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isEndOfWord);

}

int main() {

char keys[][8] = {"the", "a", "there", "answer", "any", "by", "bye", "their"}; int n = sizeof(keys) / sizeof(keys[0]);

struct TrieNode\* root = getNode();

for (int i = 0; i < n; i++) insert(root, keys[i]);

search(root, "the") ? printf("Yes\n") : printf("No\n");

search(root, "these") ? printf("Yes\n") : printf("No\n");

return 0;

}

**Common Algorithmic Patterns-**

**Traversal-**







For loops, iterators

: While loop until null

: Depth-First Search (DFS) (Preorder, Inorder, Postorder), Breadth-First Search

(BFS)

* **Graphs**: DFS, BFS

**Trees**

**Linked Lists**

**Arrays**:

**Sorting-**

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

**Searching-**

Binary Search (sorted arrays)

: Binary Search Tree search

**Graphs**: DFS, BFS

**Time Complexity Cheats-**

**Trees**

**Arrays**:







: O(1)

: O(n)

: O(log n)

: O(1) (if pointer to node is known)

: O(log n) average, O(n) worst-case

: O(log n)

: O(1) average, O(n) worst-case

**Hash Table Operations (Search, Insert, Delete)**

**Heap Operations (Insert, Delete)**

**BST Operations (Search, Insert, Delete)**

**Linked List Insertion/Deletion**

**Binary Search**

**Searching in Unsorted Array**

**Accessing Array Element**













