**1. Arrays**

**Definition:** An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

**Features:**

* Fixed size
* Homogeneous elements
* Direct access to elements via index

**Operations:**

* **Access:** O(1) time complexity
* **Insertion:** O(n) time complexity (in worst case, inserting at the beginning or middle)
* **Deletion:** O(n) time complexity (in worst case, deleting from the beginning or middle)
* **Traversal:** O(n) time complexity

**Implementation:**

* Implemented using a contiguous block of memory.

**CODE:**

#include <stdio.h>

int main() {

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

for(int i = 0; i < 5; i++) {

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

}

return 0;

}

**2. Linked Lists**

**Definition:** A linked list is a linear data structure where elements, called nodes, are linked using pointers. Each node contains data and a reference (or pointer) to the next node in the sequence.

**Types:**

* Singly Linked List
* Doubly Linked List
* Circular Linked List

**Features:**

* Dynamic size
* Ease of insertion and deletion
* Sequential access

**Operations:**

* **Access:** O(n) time complexity
* **Insertion:** O(1) time complexity (if insertion at the head)
* **Deletion:** O(1) time complexity (if deletion at the head)
* **Traversal:** O(n) time complexity

**Implementation:**

* Implemented using nodes where each node contains a data field and a pointer to the next node.

**Code:**

#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;

}

}

int main() {

struct Node\* head = NULL;

struct Node\* second = NULL;

struct Node\* third = NULL;

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

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

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

head->data = 1;

head->next = second;

second->data = 2;

second->next = third;

third->data = 3;

third->next = NULL;

printList(head);

return 0;

}

**3. Stacks**

**Definition:** A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The last element added is the first one to be removed.

**Features:**

* LIFO structure
* Dynamic size

**Operations:**

* **Push (insert):** O(1) time complexity
* **Pop (remove):** O(1) time complexity
* **Peek (access top element):** O(1) time complexity
* **IsEmpty:** O(1) time complexity

**Implementation:**

* Can be implemented using arrays or linked lists.

**Code:**

**#**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;

}

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

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

if (isFull(stack))

return;

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

}

int pop(struct Stack\* stack) {

if (isEmpty(stack))

return -1;

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

}

int peek(struct Stack\* stack) {

if (isEmpty(stack))

return -1;

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

}

int main() {

struct Stack\* stack = createStack();

push(stack, 10);

push(stack, 20);

push(stack, 30);

printf("%d popped from stack\n", pop(stack));

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

return 0;

}

**4. Queues**

**Definition:** A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The first element added is the first one to be removed.

**Types:**

* Simple Queue
* Circular Queue
* Priority Queue
* Double-ended Queue (Deque)

**Features:**

* FIFO structure
* Dynamic size

**Operations:**

* **Enqueue (insert):** O(1) time complexity
* **Dequeue (remove):** O(1) time complexity
* **Peek/Front (access first element):** O(1) time complexity
* **Rear (access last element):** O(1) time complexity
* **IsEmpty:** O(1) time complexity

**Implementation:**

* Can be implemented using arrays or linked lists.
* Circular queues use a circular buffer.

**5. Trees**

**Definition:** A tree is a hierarchical data structure consisting of nodes, with a root node and subnodes forming a parent-child relationship.

**Types:**

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

**Features:**

* Non-linear structure
* Hierarchical relationships
* Recursive definition

**Operations:**

* **Insertion:** O(log n) for balanced trees, O(n) for unbalanced trees
* **Deletion:** O(log n) for balanced trees, O(n) for unbalanced trees
* **Traversal:** O(n) time complexity for inorder, preorder, and postorder traversals
* **Search:** O(log n) for balanced trees, O(n) for unbalanced trees

**Implementation:**

* Nodes contain data and references to child nodes.

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left, \*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;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

printf("Inorder traversal: ");

inorder(root);

return 0;

}

**6. Graphs**

**Definition:** A graph is a collection of nodes (vertices) and edges (connections between nodes).

**Types:**

* Directed Graph (Digraph)
* Undirected Graph
* Weighted Graph
* Unweighted Graph

**Features:**

* Non-linear structure
* Can be cyclic or acyclic

**Operations:**

* **Add Vertex:** O(1) time complexity
* **Add Edge:** O(1) time complexity
* **Remove Vertex:** O(V + E) time complexity
* **Remove Edge:** O(1) time complexity
* **Traversal:** O(V + E) time complexity for BFS and DFS

**Implementation:**

* Adjacency Matrix: Uses O(V^2) space complexity
* Adjacency List: Uses O(V + E) space complexity

**Code:**

#include <stdio.h>

#define V 5

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, 0, 1},

{0, 1, 0, 1, 0},

{0, 0, 1, 0, 1},

{1, 1, 0, 1, 0}

};

printGraph(graph);

return 0;

}

**7. Hash Tables**

**Definition:** A hash table is a data structure that implements an associative array, a structure that can map keys to values using a hash function.

**Features:**

* Fast access, insertion, and deletion
* Handles collisions using techniques like chaining or open addressing

**Operations:**

* **Insert:** O(1) average time complexity, O(n) worst case
* **Delete:** O(1) average time complexity, O(n) worst case
* **Search:** O(1) average time complexity, O(n) worst case

**Implementation:**

* Uses an array of linked lists or a probing mechanism.

**8. Heaps**

**Definition:** A heap is a specialized tree-based data structure that satisfies the heap property (min-heap or max-heap).

**Features:**

* Complete binary tree
* Min-heap: Parent nodes are smaller than child nodes
* Max-heap: Parent nodes are larger than child nodes

**Operations:**

* **Insert:** O(log n) time complexity
* **Delete/Extract Min or Max:** O(log n) time complexity
* **Peek/Find Min or Max:** O(1) time complexity
* **Heapify:** O(n) time complexity

**Implementation:**

* Usually implemented using an array.

**9. Tries**

**Definition:** A trie is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings.

**Features:**

* Nodes represent characters of strings
* Efficient for prefix-based searches

**Operations:**

* **Insert:** O(L) time complexity, where L is the length of the key
* **Delete:** O(L) time complexity
* **Search:** O(L) time complexity

**Implementation:**

* Each node contains an array of pointers for each possible character.