**CHEATCODES OF DATASTRUCTURES**

**1. Arrays**

**Definition**: A collection of elements stored in contiguous memory locations.

Operations:

**Access:** arr[index] (O(1))

**Insertion:** Shift elements (O(n))

**Deletion:** Shift elements (O(n))

**Time Complexity:**

**Access:** O(1)

**Insertion/Deletion:** O(n)

**Space Complexity:** O(n)

**Sample Code:**

int arr[10]; // Declaration

arr[0] = 1; // Insertion

int x = arr[0]; // Access

**Applications:** Store fixed-size data, buffer storage.

**Advantages:** Fast access by index.

**Disadvantages:** Fixed size, expensive insertions and deletions.

**2.Linked Lists**

**Definition:** A collection of nodes where each node contains data and a pointer to the next node.

**Operations:**

**Insertion:** Add new node (O(1) front)

**Deletion:** Remove node (O(1) front)

**Traversal:** Visit all nodes (O(n))

**Time Complexity:**

**Insertion/Deletion:** O(1) (front)

**Traversal:** O(n)

**Space Complexity:** O(n)

**Sample code:**

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

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

newNode->data = 1;

newNode->next = head;

head = newNode; // Insertion at the front

**Applications:** Dynamic memory allocation, stack and queue implementation.

**Advantages:** Dynamic size, easy insertion/deletion.

**Disadvantages:** Extra memory for pointers, linear access time.

**3.Stacks**

**Definition:** A collection of elements with Last In First Out (LIFO) access.

**Operations:**

**Push:** Add element to top (O(1))

**Pop:** Remove element from top (O(1))

**Peek:** View top element (O(1))

**Time Complexity:** O(1) for all operations.

**Space Complexity:** O(n)

**Sample Code:**

#define MAX 100

int stack[MAX], top = -1;

void push(int x) { stack[++top] = x; }

int pop() { return stack[top--]; }

int peek() { return stack[top]; }

**Applications:** Function call management, expression evaluation.

**Advantages:** Simple implementation, efficient access to the top element.

**Disadvantages:** Limited access to only the top element.

**4.Queues**

**Definition:** A collection of elements with First In First Out (FIFO) access.

**Operations:**

**Enqueue:** Add element to end (O(1))

**Dequeue:** Remove element from front (O(1))

**Peek:** View front element (O(1))

**Time Complexity:** O(1) for all operations.

**Space Complexity:** O(n)

**Sample Code:**

#define MAX 100

int queue[MAX], front = -1, rear = -1;

void enqueue(int x) { queue[++rear] = x; if (front == -1) front = 0; }

int dequeue() { int x = queue[front]; if (front == rear) front = rear = -1; else front++; return x; }

int peek() { return queue[front]; }

**Applications:** Process scheduling, buffering.

**Advantages:** Simple to implement, efficient for FIFO operations.

**Disadvantages:** Fixed size (array implementation), less flexible than linked list-based implementation.

**5.Binary Trees**

**Definition:** A hierarchical data structure in which each node has at most two children.

**Operations:**

**Insertion:** Add node (O(n))

**Deletion:** Remove node (O(n))

**Search:** Find node (O(n))

**Traversal:** Visit nodes (O(n))

**Time Complexity:** O(n) for insertion, deletion, search, and traversal.

**Space Complexity:** O(n)

**Sample Code:**

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;

}

**Applications:** Hierarchical data representation, parsing expressions.

**Advantages:** Flexible size, straightforward traversal algorithms.

**Disadvantages:** Can become unbalanced, leading to poor performance.

**6.Binary Search Trees (BST)**

**Definition:** A binary tree where each node has a key greater than all the keys in the left subtree and less than those in the right subtree.

**Operations:**

**Insertion:** Add node (O(log n) avg)

**Deletion:** Remove node (O(log n) avg)

**Search:** Find node (O(log n) avg)

**Time Complexity:** O(log n) avg, O(n) worst case.

**Space Complexity:** O(n)

**Sample Code:**

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

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

return node;

}

**Applications:** Efficient searching and sorting operations.

**Advantages:** Efficient searching, dynamic size.

**Disadvantages:** Can become unbalanced, leading to O(n) operations.

**7.AVL Trees**

**Definition:** A self-balancing binary search tree where the height of the left and right subtrees of every node differ by at most one.

**Operations:**

**Insertion:** Add node (O(log n))

**Deletion:** Remove node (O(log n))

**Search:** Find node (O(log n))

**Time Complexity:** O(log n) for all operations.

**Space Complexity:** O(n)

**Sample Code:**

struct Node {

int data;

struct Node\* left;

struct Node\* right;

int height;

};

int height(struct Node\* n) { return n ? n->height : 0; }

struct Node\* rightRotate(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

// Similar leftRotate and balance functions needed

**Applications:** Databases, file systems where frequent insertions and deletions occur.

**Advantages:** Always balanced, guaranteed O(log n) operations.

**Disadvantages:** More complex rotations, higher constant factors due to balancing overhead.

**8.Graphs**

**Definition:** A set of vertices connected by edges.

**Operations:**

**Add Edge:** Connect vertices (O(1) for adjacency list)

**Remove Edge:** Disconnect vertices (O(1) for adjacency list)

**Search:** BFS/DFS (O(V + E))

**Time Complexity:**

**Add/Remove Edge:** O(1) for adjacency list

**Search:** O(V + E)

**Space Complexity:**

**Adjacency List:** O(V + E)

**Adjacency Matrix:** O(V^2)

**Sample Code:**

struct Graph {

int V;

struct Node\* adjList[];

};

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

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

newNode->dest = dest;

newNode->next = graph->adjList[src];

graph->adjList[src] = newNode;

}

**Applications:** Network routing, social networks, pathfinding algorithms.

**Advantages:** Flexible representation, suitable for complex relationships.

**Disadvantages:** Can be complex to implement and manage.

**9.Hashing**

**Definition**: A technique to map keys to values using a hash function.

**Operations:**

**Insert:** Add key-value pair (O(1) avg)

**Delete:** Remove key-value pair (O(1) avg)

**Search:** Find value by key (O(1) avg)

**Time Complexity:** O(1) average for all operations.

**Space Complexity:** O(n)

**Sample Code:**

#define TABLE\_SIZE 100

struct Node {

int key;

int value;

struct Node\* next;

};

struct Node\* hashTable[TABLE\_SIZE];

int hashFunction(int key) { return key % TABLE\_SIZE; }

void insert(int key, int value) {

int hashIndex = hashFunction(key);

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

newNode->key = key;

newNode->value = value;

newNode->next = hashTable[hashIndex];

hashTable[hashIndex] = newNode;

}

**Applications:** Efficient data retrieval, databases, caching.

**Advantages:** Fast data retrieval.

**Disadvantages:** Hash collisions, poor performance with bad hash functions or high load factors.