1. **Arrays**:
   * **Definition**: An array is a collection of elements stored in contiguous memory locations, where each element is accessed by its index.
   * **Operations**:
     + Access: O(1)
     + Insertion (at the end): O(1)
     + Deletion (at the end): O(1)
     + Search (Unsorted): O(n)
     + Search (Sorted): O(log n) - for binary search
   * **Types**: One-dimensional, Multi-dimensional
   * **Shorter Code**:

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

printf("%d", arr[2]); // Accessing element at index 2

// Accessing element at index 2

1. **Linked Lists**:
   * **Definition**: A linked list is a collection of nodes where each node contains data and a reference (link) to the next node in the sequence.
   * **Operations**:
     + Insertion: O(1) - at the beginning or end, O(n) - in the middle
     + Deletion: O(1) - if the node is known, O(n) - if the node needs to be found
     + Search: O(n)
   * **Types**: Singly Linked List, Doubly Linked List, Circular Linked List
   * **Shorter Code:**

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

// Inserting a node at the beginning

void insert(int data) {

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

newNode->data = data;

newNode->next = head;

head = newNode;

}

1. **Stacks**:
   * **Definition**: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.
   * **Operations**:
     + Push: O(1)
     + Pop: O(1)
     + Peek: O(1)
   * **Types**: Array-based stack, Linked list-based stack
   * **Shorter Code**:

#define MAX 1000

struct Stack {

int top;

int array[MAX];

};

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

if (stack->top == MAX - 1) return;

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

}

int pop(struct Stack\* stack) {

if (stack->top == -1) return INT\_MIN;

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

}

4. **Queues:**

* + **Definition**: A queue is a linear data structure that follows the First In, First Out (FIFO) principle.
  + **Operations**:
    - Enqueue: O(1)
    - Dequeue: O(1)
    - Peek: O(1)
  + **Types**: Array-based queue, Linked list-based queue
  + **Shorter Code**:

#define MAX 1000

struct Queue {

int front, rear, size;

unsigned capacity;

int array[MAX];

};

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

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

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

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

queue->size++;

}

int dequeue(struct Queue\* queue) {

if (queue->size == 0) return INT\_MIN;

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

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

queue->size--;

return item;

}

1. **Trees**:
   * **Definition**: A tree is a hierarchical data structure consisting of nodes connected by edges.
   * **Operations**:
     + Insertion: O(log n) - for balanced trees like AVL or Red-Black trees, O(n) - for unbalanced trees
     + Deletion: O(log n) - for balanced trees, O(n) - for unbalanced trees
     + Search: O(log n) - for balanced trees, O(n) - for unbalanced trees
   * **Types**: Binary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, etc.
   * **Binary Tree:**
   * **Shorter Code**:

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);

}

}

6**. Hash Table:**

**Definition:**

A data structure that maps keys to values for efficient lookup.

**Operations:**

Insert: O(1)

Delete: O(1)

Search: O(1)

**Shorter Code**:

#define SIZE 20

struct HashItem {

int key;

int value;

};

struct HashItem\* hashArray[SIZE];

int hashCode(int key) {

return key % SIZE;

}

void insert(int key, int value) {

struct HashItem\* item = (struct HashItem\*) malloc(sizeof(struct HashItem));

item->key = key;

item->value = value;

int hashIndex = hashCode(key);

while (hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

hashIndex++;

hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

struct HashItem\* search(int key) {

int hashIndex = hashCode(key);

while (hashArray[hashIndex] != NULL) {

if (hashArray[hashIndex]->key == key)

return hashArray[hashIndex];

hashIndex++;

hashIndex %= SIZE;

}

return NULL;

}

7**. Graph:**

**Definition:**

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

**Operations:**

Add Vertex: O(1)

Add Edge: O(1)

Remove Vertex: O(V+E)

Remove Edge: O(E)

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

**Shorter Code**:

struct Node {

int dest;

struct Node\* next;

};

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;

}

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

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

newNode->dest = dest;

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

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

}