Different Types Of Data Structures

1. Arrays

Definition: An array is a collection of elements, all of the same type, stored in contiguous memory locations.

Operations:

- **Access (Read/Write)**:
    - Description: Accessing elements by index.
    - Complexity: O(1) - Constant time complexity.
- **Search**:
    - Description: Searching for a specific element in the array.
    - Complexity: O(n) - Linear time complexity for unsorted arrays. O(log n) if the array is sorted and binary search is used.
- **Insertion**:
    - Description: Adding an element at a specific position in the array.
    - Complexity: O(n) - Linear time complexity. O(1) if inserting at the end.
- **Deletion**:
    - Description: Removing an element from a specific position in the array.
    - Complexity: O(n) - Linear time complexity. O(1) if deleting from the end.

Space Complexity: O(n)

Advantages:

- Simple and easy to use.
- Fast access to elements.

Disadvantages:

- Fixed size, cannot grow or shrink dynamically.
- Insertion and deletion are expensive (linear time complexity).

Applications:

- Storing collections of data.
- Used in sorting algorithms like bubble sort, quick sort, etc.

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 each element, called a node, contains data and a reference (link) to the next node in the sequence.

Operations:

- **Access (Read/Write)**:
    - Description: Accessing elements sequentially by traversing the linked list.
    - Complexity: O(n) - Linear time complexity.
- **Search**:
    - Description: Searching for a specific element in the linked list.
    - Complexity: O(n) - Linear time complexity.
- **Insertion**:
    - Description: Adding an element at the beginning, end, or middle of the linked list.
    - Complexity: O(1) - Constant time complexity for insertion at the beginning. O(n) for insertion at the end or middle.
- **Deletion**:
    - Description: Removing an element from the linked list.
    - Complexity: O(1) - Constant time complexity if removing from the beginning. O(n) for removal from the end or middle.

Space Complexity: O(n)

Advantages:

- Dynamic size.
- Efficient insertion and deletion compared to arrays.

Disadvantages:

- No random access to elements.
- Extra memory required for storing pointers.

Applications:

- Dynamic memory management.
- Implementing stacks and queues.

Code:

```
#include <stdio.h>

#include <stdlib.h>

struct Node { int data; struct Node* next; };

int main() {

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

    head->data = 1; head->next = NULL;

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

    free(head);

    return 0;

}
```

3. Stacks

Definition: A stack is a linear data structure that follows the LIFO (Last In First Out) principle. The operations are performed at one end, called the top.

Operations:

- **Push**:
  - Description: Adding an element to the top of the stack.
  - Complexity: O(1) - Constant time complexity.
- **Pop**:
  - Description: Removing the top element from the stack.
  - Complexity: O(1) - Constant time complexity.
- **Peek**:
  - Description: Viewing the top element of the stack without removing it.
  - Complexity: O(1) - Constant time complexity.

Space Complexity: O(n)

Advantages:

- Simple and efficient for adding/removing elements.

Disadvantages:

- Limited access to elements (only the top element can be accessed directly).

Applications:

- Expression evaluation (e.g., parsing arithmetic expressions).

- Function call management (call stack in programming languages).

Code:

```
#include <stdio.h>

#define MAX 100

struct Stack { int arr[MAX]; int top; };

void main() {

   struct Stack stack = { .top = -1 };

   stack.arr[++stack.top] = 10;

   printf("%d ", stack.arr[stack.top--]);

}
```

4. Queues

Definition: A queue is a linear data structure that follows the FIFO (First In First Out) principle. Elements are added at the rear and removed from the front.

Operations:

- **Enqueue**:
  - Description: Adding an element to the rear of the queue.
  - Complexity: O(1) - Constant time complexity.
- **Dequeue**:
  - Description: Removing the front element from the queue.
  - Complexity: O(1) - Constant time complexity.
- **Peek**:
  - Description: Viewing the front element of the queue without removing it.
  - Complexity: O(1) - Constant time complexity.

Space Complexity: O(n)

Advantages:

- Simple and efficient for adding/removing elements in FIFO order.

Disadvantages:

- Limited access to elements (only the front and rear elements can be accessed directly).

Applications:

- Task scheduling (e.g., print queues, CPU scheduling).
- Breadth-First Search (BFS) in graph algorithms.

Code:

```
#include <stdio.h>

#define MAX 100

struct Queue { int arr[MAX]; int front, rear; };

void main() {

   struct Queue queue = { .front = -1, .rear = -1 };

   queue.arr[++queue.rear] = 10;

   printf("%d ", queue.arr[queue.front++]);

}
```

5. Trees

Definition: A tree is a hierarchical data structure consisting of nodes, with a single node designated as the root. Each node has zero or more child nodes.

Operations:

- **Insertion**:
  - Description: Adding a new node to the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Deletion**:
  - Description: Removing a node from the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Traversal**:
  - Description: Visiting all nodes of the tree in a specific order.
  - Complexity: O(n) - Linear time complexity for visiting all nodes.

Space Complexity: O(n)

Types:

- Binary trees: Each node has at most two children.
- Binary search trees (BST): Binary tree with ordered nodes.
- AVL trees: Self-balancing binary search tree.
- Heaps: Complete binary tree used for priority queues.

Advantages:

- Efficient search, insertion, and deletion (depending on the tree type).

Disadvantages:

- Can become unbalanced, leading to inefficiencies (except for self-balancing trees).

Applications:

- Hierarchical data representation (e.g., file systems).
- Searching and sorting algorithms (e.g., binary search tree, heapsort).

6. Graphs

Definition: A graph is a collection of nodes (vertices) and edges connecting pairs of nodes. Graphs can be directed or undirected.

Operations:

- **Insertion**:
  - Description: Adding a new node to the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Deletion**:
  - Description: Removing a node from the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Traversal**:
  - Description: Visiting all nodes of the tree in a specific order.
  - Complexity: O(n) - Linear time complexity for visiting all nodes.

Space Complexity: O(V + E)

Types:

- Directed graphs: Edges have a direction.
- Undirected graphs: Edges do not have a direction.
- Weighted graphs: Edges have weights.

Advantages:

- Flexible representation of relationships.

Disadvantages:

- Can be complex to implement and understand.

Applications:

- Network routing algorithms (e.g., Dijkstra's algorithm).
- Social network analysis.
- Representation of dependencies (e.g., task scheduling).

7. Hash Tables

Definition: A hash table is a data structure that maps keys to values using a hash function to compute an index into an array of buckets.

Operations:

- **Insertion**:
  - Description: Adding a new node to the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Deletion**:
  - Description: Removing a node from the tree.
  - Complexity: O(log n) - Logarithmic time complexity for balanced trees.
- **Traversal**:
  - Description: Visiting all nodes of the tree in a specific order.
  - Complexity: O(n) - Linear time complexity for visiting all nodes.

Space Complexity: O(n) (depends on the number of elements stored and the size of the hash table)

Advantages:

- Fast access to elements (constant average time complexity).

Disadvantages:

- Potential for collisions (two keys mapping to the same index).
- Extra space for hash table maintenance.

Applications:

- Implementing associative arrays (dictionaries).
- Caching and indexing for fast data retrieval.

8. Heaps

Definition: A heap is a specialized tree-based data structure that satisfies the heap property: for a max-heap, the key of each node is greater than or equal to the keys of its children; for a min-heap, the key of each node is less than or equal to the keys of its children.

Heap Operations:

1. **Heapify**
   - **Definition**: Convert an array into a heap.
   - **Complexity**: O(n) - Linear time complexity, where n is the number of elements in the array.
2. **Insertion**
   - **Definition**: Add a new element to the heap while maintaining the heap property.
   - **Complexity**: O(log n) - Logarithmic time complexity, where n is the number of elements in the heap.
3. **Deletion (Extract Max or Min)**

    - **Definition**: Remove the root element from the heap while maintaining the heap property.
    - **Complexity**: O(log n) - Logarithmic time complexity, where n is the number of elements in the heap.
4. **Heapify Up (Bubble-Up)**
    - **Definition**: Restore heap property after insertion by moving the newly inserted element up the tree as necessary.
    - **Complexity**: O(log n) - Logarithmic time complexity, where n is the number of elements in the heap.
5. **Heapify Down (Bubble-Down)**
    - **Definition**: Restore heap property after deletion by moving the replacement element down the tree as necessary.
    - **Complexity**: O(log n) - Logarithmic time complexity, where n is the number of elements in the heap.
6. **Heapify Subtree**
    - **Definition**: Heapify a subtree rooted at a given node.
    - **Complexity**: O(log n) - Logarithmic time complexity, where n is the number of elements in the heap.

Space Complexity: O(n)

Types:

- Min-heap: Smallest element is at the root.
- Max-heap: Largest element is at the root.

Advantages:

- Efficient for priority queue operations (insert, extract-min/max).

Disadvantages:

- Limited access to elements (only the root element can be accessed directly).

Applications:

- Priority queues.
- Heap sort algorithm.

CONCLUSION:

Each data structure serves a specific purpose and is chosen based on the requirements of the problem at hand. Understanding their characteristics, operations, and use cases helps in selecting the appropriate data structure for efficient problem solving.