Data Structure Classification

Data structures are classified based on how data is organized in memory and how elements relate to each other.

1. Linear Data Structures

Linear data structures store data elements sequentially. Each element is connected to its previous and next element.

a) Static Linear Data Structures

Static data structures have a fixed size defined at compile time. Memory allocation cannot be changed during execution.

- Arrays

Why used:

- Fast random access using index
- Efficient memory usage for fixed-size data

Applications:

- Image processing (pixel matrices)
- Storing fixed sensor readings
- Lookup tables

b) Dynamic Linear Data Structures

Dynamic structures can grow or shrink during program execution. Memory is allocated at runtime.

- Linked Lists
- Stacks
- Queues

Why used:

- Efficient insertion and deletion
- No wasted memory due to fixed size

Applications:

- Undo/Redo systems (Stacks)
- CPU scheduling (Queues)
- Dynamic memory management

2. Non-Linear Data Structures

Non-linear data structures store data hierarchically or graph-like, allowing one-to-many relationships.

a) Trees

Trees consist of nodes connected in a hierarchical manner. Each tree has a root and child nodes.

Why used:

- Efficient searching and sorting
- Represents hierarchical data naturally

Applications:

- File systems
- Database indexing (B-Trees)
- DOM structure in web browsers

b) Graphs

Graphs consist of vertices (nodes) connected by edges. They can be directed or undirected.

Why used:

- Models real-world relationships
- Supports complex connections

Applications:

- Social networks
- Routing algorithms
- Maps and navigation systems

Applications of Data Structures and Algorithms

Data structures are chosen based on the problem requirements and the algorithm operating on them.

Arrays + Binary Search

Binary search requires a sorted array to work efficiently.

Used in:

- Databases
- Search engines
- Operating system resource lookup

Reason:

- O(log n) search time
- Predictable memory layout

Stacks + Recursion

Function calls are managed using stacks.

Used in:

- Program execution
- Expression evaluation
- Undo/Redo features

Reason:

- Last function call must return first

Queues + Scheduling Algorithms

Queues are used where tasks must be processed in order.

Used in:

- CPU scheduling
- Printer queues
- Network packet handling

Reason:

- Fairness
- Predictable processing order

How Data Structures and Algorithms Work Within Systems

Data structures and algorithms form the foundation of all computer systems. They determine system performance, scalability, and reliability.

Operating Systems

- Queues manage process scheduling
- Stacks manage function calls and interrupts
- Linked lists track memory blocks

Databases

- Trees index large datasets
- Hash tables provide fast lookups
- Graphs represent relationships

Networking Systems

- Graphs determine routing paths
- Queues manage packet transmission

Software Applications

- Text editors use stacks for undo operations
- Games use trees for scene management
- Web browsers use trees for HTML rendering

Efficient data structures combined with suitable algorithms ensure systems are fast, scalable, and resource-efficient.

Arrays

An **array** stores elements in contiguous memory locations. This allows constant-time access using an index.

Example

```
#include <stdio.h>

int main() {
    int arr[4] = {10, 20, 30, 40};

    printf("%d\n", arr[0]); // 10
    printf("%d\n", arr[2]); // 30

    return 0;
}
```

Why insertion is expensive

To insert an element in the middle, all following elements must be shifted.

Time Complexity

Operation	Time
Access	O(1)
Search	O(n)
Insert / Delete	O(n)

Linked Lists

A **linked list** is made of nodes where each node contains data and a pointer to the next node.

Node Definition

```
#include <stdlib.h>

struct Node {
    int data;
```

```
    struct Node* next;
};
```

Create and Link Nodes

```
struct Node* head = malloc(sizeof(struct Node));
head->data = 10;
head->next = malloc(sizeof(struct Node));
head->next->data = 20;
head->next->next = NULL;
```

Linked lists trade fast access for fast insertion and deletion.

Stacks

A **stack** follows the **LIFO** principle (Last In, First Out).

Array-Based Stack

```
#define MAX 5
int stack[MAX];
int top = -1;

void push(int value) {
    if (top == MAX - 1) return;
    stack[++top] = value;
}

int pop() {
    if (top == -1) return -1;
    return stack[top--];
}
```

Used in function calls, recursion, undo/redo operations.

Queues

A **queue** follows the **FIFO** principle (First In, First Out).

Simple Queue Implementation

```
#define MAX 5
int queue[MAX];
int front = 0, rear = -1;

void enqueue(int value) {
    if (rear == MAX - 1) return;
    queue[++rear] = value;
}

int dequeue() {
    if (front > rear) return -1;
    return queue[front++];
}
```

Queues are essential for BFS and scheduling algorithms.

Trees

A **tree** is a hierarchical data structure made of nodes.

Binary Tree Node

```
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

Create a Node

```
struct TreeNode* newNode(int value) {
    struct TreeNode* node = malloc(sizeof(struct TreeNode));
    node->data = value;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

Algorithms

Linear Search

Checks each element until the target is found.

```
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Time Complexity: O(n)

Binary Search

Works on sorted arrays by repeatedly halving the search space.

```
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Time Complexity: O(log n)