All Data Structures in Programming

Data structures are fundamental for organizing and managing data efficiently in programming. They can be classified into **linear** and **non-linear** data structures. Below is a comprehensive list:

---

1. Linear Data Structures

- Data is arranged sequentially.
- Elements are stored in contiguous memory locations.

a) Arrays

- Fixed-size collection of elements of the same type.
- Random access is possible.

Example:
java
CopyEdit
```java
int[] arr = {10, 20, 30, 40};
```

-

b) Linked Lists

- Consists of nodes where each node contains data and a reference to the next node.
- Types:
  - **Singly Linked List** (each node points to the next)
  - **Doubly Linked List** (each node has references to both previous and next nodes)
  - **Circular Linked List** (last node points back to the first node)

Example:
java
CopyEdit
```java
class Node {
    int data;
    Node next;
}
```

-

c) Stacks (LIFO – Last In, First Out)

- Elements are added and removed from the same end (top).
- Operations:
  - **Push** (add element)
  - **Pop** (remove element)
  - **Peek** (get top element)

Example:
java
CopyEdit
```java
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.pop();
```

-

d) Queues (FIFO – First In, First Out)

- Elements are added at the **rear** and removed from the **front**.
- Types:
  - **Simple Queue** (FIFO)
  - **Circular Queue** (last connects to first)
  - **Deque (Double-Ended Queue)** (insert/remove from both ends)
  - **Priority Queue** (elements are dequeued based on priority)

Example:
java
CopyEdit
```java
Queue<Integer> queue = new LinkedList<>();
queue.add(10);
queue.poll();
```

-

---

2. Non-Linear Data Structures

- Data is arranged hierarchically or connected in complex relationships.

a) Trees

- Hierarchical data structure.

- Types:
    - **Binary Tree** (each node has ≤2 children)
    - **Binary Search Tree (BST)** (left < root < right)
    - **AVL Tree** (self-balancing BST)
    - **B-Trees & B+ Trees** (used in databases)
    - **Heap** (Min-Heap, Max-Heap)

Example:
java
CopyEdit
```java
class Node {
    int data;
    Node left, right;
}
```

-

b) Graphs

- Collection of **nodes (vertices)** connected by **edges**.
- Types:
    - **Directed Graph (edges have direction)**
    - **Undirected Graph (edges are bidirectional)**
    - **Weighted Graph (edges have weights)**
    - **Unweighted Graph**
- Representation:
    - **Adjacency Matrix**
    - **Adjacency List**

Example:
java
CopyEdit
```java
List<List<Integer>> graph = new ArrayList<>();
```

-

---

3. Hash-Based Data Structures

- Used for fast lookups (key-value pairs).

a) Hash Tables / Hash Maps

- Stores key-value pairs.
- Uses **hash functions** to map keys to indices.

Example:
java
CopyEdit
```
HashMap<String, Integer> map = new HashMap<>();
map.put("Alice", 25);
```

-

b) Hash Sets

- Similar to HashMap but stores only unique values.

Example:
java
CopyEdit
```
HashSet<Integer> set = new HashSet<>();
set.add(10);
```

-

---

4. Special Data Structures

- Used for advanced operations.

a) Tries (Prefix Trees)

- Used for searching words efficiently.

Example:
java
CopyEdit
```
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEndOfWord;
}
```

-

b) Bloom Filters

- Used for fast membership testing with a small memory footprint.

c) Skip Lists

- Alternative to balanced trees for faster search.

d) Disjoint Set (Union-Find)

- Used in graph algorithms like Kruskal's MST.

Conclusion

Each data structure has a specific use case depending on the efficiency required for operations like searching, inserting, and deleting data. Understanding when to use **arrays, linked lists, trees, graphs, stacks, queues, hash tables, and tries** is crucial in programming.