CodInClub
Powered By BridgeLabz

Data Structures Overview

A **data structure** is a way of organizing and storing data so it can efficiently perform various operations such as searching, sorting, and updating. They are essential for optimizing algorithms and managing data effectively in applications.

Examples include **Arrays**, **Stacks**, **Queues**, **Linked Lists**, **Trees**, and **Graphs**. Among these, **Linked Lists** are particularly important for dynamic memory usage.

Key Features of Data Structures in Java

1. **Predefined and Custom Implementations**:
   - Java provides built-in data structures (e.g., Arrays, Lists, Maps) in the **Java Collections Framework**.
   - Custom data structures can be implemented using classes and objects.
2. **Dynamic Memory Allocation**:
   - Some structures like Lists and Maps can grow or shrink dynamically, making them more flexible than fixed-size arrays.
3. **Generics Support**:
   - Data structures in Java are type-safe, allowing developers to specify the type of elements stored, reducing runtime errors.
4. **Thread-Safe Options**:
   - Java provides concurrent data structures (e.g., `ConcurrentHashMap`, `CopyOnWriteArrayList`) for multi-threaded environments.

Types of Data Structures in Java

1. **Linear Data Structures**:
   - **Array**: A fixed-size collection of elements stored in contiguous memory locations.
   - **Linked List**: A sequence of nodes where each node contains data and a reference to the next node.
   - **Stack**: A LIFO (Last-In-First-Out) structure for managing elements.
   - **Queue**: A FIFO (First-In-First-Out) structure, often used in scheduling tasks.
2. **Hierarchical Data Structures**:
   - **Tree**: A collection of nodes organized in a hierarchy, starting from a root node.
   - **Binary Tree**: A tree where each node has at most two children.
   - **Binary Search Tree (BST)**: A binary tree with nodes organized for quick searches.
   - **Heap**: A special tree-based structure for priority-based tasks.

3. **Graph**:
    - A collection of nodes (vertices) and edges where relationships are represented.
    - Can be **directed** or **undirected**.
4. **Hash-Based Data Structures**:
    - **HashMap**: Stores key-value pairs for efficient retrieval.
    - **HashSet**: Stores unique elements for quick membership checks.
5. **Advanced Data Structures**:
    - **Trie**: Used for prefix-based searching.
    - **Segment Tree**: Used for range queries.
    - **Red-Black Tree**: A self-balancing binary search tree.

Linked List

A **linked list** is a linear data structure where elements (called nodes) are linked using pointers. Each node consists of two parts:

1. **Data**: Holds the value of the node.
2. **Pointer**: Points to the next node (or previous node, in the case of doubly linked lists).

---

1. Singly Linked List (SLL)

In a **Singly Linked List**, each node contains data and a pointer to the next node. The last node points to `null`.

Example Structure:

```
Head -> Node1 -> Node2 -> Node3 -> null
```

Operations:

1. **Insertion**:
    - At the beginning
    - At the end
    - At a specific position
2. **Deletion**:
    - From the beginning
    - From the end
    - From a specific position
3. **Traversal**:

- Visit each node sequentially.

Advantages:

- Dynamic size: Memory is allocated as needed.
- Efficient insertion and deletion compared to arrays.

Disadvantages:

- Sequential access only (no direct access to elements).
- More memory usage due to pointers.

---

2. Doubly Linked List (DLL)

In a **Doubly Linked List**, each node contains:

- Data.
- A pointer to the next node.
- A pointer to the previous node.

Example Structure:

```
null <- Node1 <-> Node2 <-> Node3 -> null
```

Operations:

1. **Insertion**:
   - At the beginning: Adjust `prev` pointer of the head.
   - At the end: Adjust the `next` pointer of the last node.
   - At a specific position: Adjust both `prev` and `next` pointers.
2. **Deletion**:
   - Update both `prev` and `next` pointers.
3. **Traversal**:
   - Forward traversal: From head to tail.
   - Backward traversal: From tail to head.

Advantages:

- Traversal in both directions.
- Easier to delete a node when a reference to it is provided.

Disadvantages:

- Requires more memory due to two pointers per node.

---

3. Circular Linked List (CLL)

In a **Circular Linked List**, the last node points back to the first node, forming a circle.

Example Structure:

a. Singly Circular Linked List:

```
Node1 -> Node2 -> Node3 -> Node1 (Head)
```

b. Doubly Circular Linked List:

```
Node1 <-> Node2 <-> Node3 <-> Node1 (Head)
```

Operations:

1. **Insertion**:
   - At the beginning: Update the last node to point to the new head.
   - At the end: Update the new node to point to the head.
2. **Deletion**:
   - Update the links to skip the node being deleted.
3. **Traversal**:
   - Use a loop to traverse until the head is encountered again.

Advantages:

- Suitable for circular operations (e.g., scheduling, buffering).
- No `null` values, making traversal simpler in certain cases.

Disadvantages:

- More complex to implement.
- Requires careful handling to avoid infinite loops.

Example Use Cases

Singly Linked List:

- Representing a sequence of elements dynamically (e.g., students in a class).
- Example:

```
class Node {
```

```
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

Doubly Linked List:

- Browser history: Backward and forward navigation.
- Example:

```
class Node {
    int data;
    Node next, prev;

    public Node(int data) {
        this.data = data;
        this.next = this.prev = null;
    }
}
```

Circular Linked List:

- Implementing a Round-Robin scheduler.
- Example:

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = this; // Pointing to itself for circularity
    }
}
```

Comparison Table

Feature	Singly Linked List	Doubly Linked List	Circular Linked List
Memory	Less (one pointer)	More (two pointers)	Similar to SLL/DLL
Traversal	Forward only	Forward and backward	Circular traversal
Complexity	Simpler implementation	More complex due to `prev`	Most complex due to circularity
Use Case	Basic dynamic data storage	Complex data relationships	Cyclical operations

Programming Concept Practiced

1. Employee Record Management Using Linked Lists

Objective: Manage employee records using a **Singly Linked List**. Each node in the list represents an employee record with fields like `ID`, `Name`, `Department`, and `Salary`.

Features:

1. **Add Employee**: Insert a new employee record at the beginning, end, or a specific position.
2. **Delete Employee**: Remove an employee record by `ID`.
3. **Search Employee**: Find an employee record by `ID` or `Name`.
4. **Display Records**: Display all employee records in the linked list.

Structure of a Node:

```
class EmployeeNode {
```

```
    int id;
    String name;
    String department;
    double salary;
    EmployeeNode next;

    public EmployeeNode(int id, String name, String department, double
salary) {
        this.id = id;
        this.name = name;
        this.department = department;
        this.salary = salary;
        this.next = null;
    }
}
```

Operations:

1. **Add Employee**:
    - If adding at the beginning, update the head of the list.
    - If adding at the end, traverse to the last node and update its `next` pointer.
2. **Delete Employee**:
    - Locate the node to be deleted by matching the `ID`.
    - Update the `next` pointer of the preceding node to skip the deleted node.
3. **Search Employee**:
    - Traverse the list, checking each node's `ID` or `Name` field.
4. **Display Records**:
    - Traverse the list from the head to the end, printing each node's data.

Use Cases:

- Manage dynamic employee data for small organizations.
- Useful for operations like adding, deleting, or searching employees without predefined limits.

---

2. Playlist Management Using Circular Linked List

CodInClub

Powered By BridgeLabz

Objective: Manage a music playlist using a **Circular Linked List**. Each node represents a song with fields like `Song Name`, `Artist`, and `Duration`.

Features:

1. **Add Song**: Insert a new song at the beginning, end, or a specific position in the playlist.
2. **Delete Song**: Remove a song by its name.
3. **Play Next Song**: Move to the next song in the playlist.
4. **Play Previous Song**: (If using a doubly circular linked list.)
5. **Display Playlist**: Print all songs in the playlist.

Structure of a Node:

```
class SongNode {
    String songName;
    String artist;
    double duration; // In minutes
    SongNode next;

    public SongNode(String songName, String artist, double duration) {
        this.songName = songName;
        this.artist = artist;
        this.duration = duration;
        this.next = this; // Point to itself for circularity
    }
}
```

Operations:

1. **Add Song**:
    - If adding at the beginning, update the new node's `next` pointer to the current head and the last node's `next` pointer to the new node.
    - If adding at the end, link the last node's `next` pointer to the new node, and the new node's `next` pointer to the head.
2. **Delete Song**:
    - Locate the node containing the song name.
    - Update the preceding node's `next` pointer to skip the deleted node.
3. **Play Next Song**:

    - Move to the `next` node from the current node.
4. **Display Playlist**:
    - Start from the head and traverse the list until the head is encountered again.

Use Cases:

- Create dynamic music playlists.
- Useful for cyclic playback without manual restarting.