Data Structure and Algorithm BCA 3rd

Topic:

Introduction to Data Structure

Introduction to Data Structure (2 hrs)

Data structures are a fundamental aspect of computer science and software engineering. They form the backbone of efficient algorithms and play a significant role in organizing and storing data for quick and efficient access and modification. Understanding data structures is crucial for anyone learning to write efficient code, as choosing the right data structure for a given problem can have a profound impact on performance and scalability.

---

1. Definition of Data Structure

A **data structure** is a specialized format for organizing, processing, retrieving, and storing data. It provides a way to manage and store data efficiently so that operations like searching, insertion, deletion, and updating can be performed with minimal time and space complexity.

In simpler terms, a data structure defines how data is stored in a computer's memory, and how it can be accessed or modified in a structured way. The choice of data structure can impact the efficiency of the algorithm being used, especially when dealing with large datasets.

Examples of Data Structures

- **Array**: A collection of elements identified by index or key.
- **Linked List**: A linear collection of nodes, each containing data and a reference to the next node.
- **Stack**: A linear collection of elements following the Last In First Out (LIFO) principle.
- **Queue**: A linear collection of elements following the First In First Out (FIFO) principle.
- **Tree**: A hierarchical data structure consisting of nodes with a parent-child relationship.
- **Graph**: A collection of nodes connected by edges, used to represent relationships or networks.

Each of these data structures serves a specific purpose and can be used in different contexts depending on the problem at hand.

---

2. Abstract Data Type (ADT)

An **Abstract Data Type (ADT)** is a theoretical concept used to define a data structure in terms of its behavior (what operations are possible) rather than its concrete implementation. An ADT focuses on what the data structure is supposed to do, not how it is implemented. This abstraction helps in designing software that is independent of the underlying implementation.

Key Characteristics of ADTs:

- **Encapsulation**: ADTs abstract away the details of how data is stored and focus only on the operations that can be performed on the data.
- **Operations**: An ADT is defined by a set of operations (e.g., insert, delete, retrieve) that are allowed on the data.
- **Interface**: ADTs specify the interface (the "what" is provided) without revealing the internal workings (the "how").

For example:

- A **Stack** is an ADT, and it provides operations like push(), pop(), and peek(). The underlying implementation might use an array or a linked list, but the user of the stack does not need to know the details of the implementation, just the operations it supports.
- A **Queue** is also an ADT, and it provides operations like enqueue() and dequeue().

In essence, ADTs are used to define the functionality and expected behaviors of data structures without getting bogged down in implementation details.

---

3. Importance of Data Structures

The importance of data structures cannot be overstated as they form the foundation of algorithms and are crucial for various applications in software development. Here are some reasons why data structures are important:

1. Efficient Data Storage and Access:

Data structures provide an efficient way of storing and accessing data. For example, searching for an element in an unsorted array may take O(n) time, but using a binary search tree can reduce the time complexity to O(log n), making a significant performance improvement when dealing with large datasets.

2. Optimal Use of Resources:

Choosing the right data structure can make the system more efficient in terms of both **time** and **space**. For instance, an array might be the best choice when elements are fixed and frequently accessed, while a linked list could be more suitable when data needs to be dynamically inserted or deleted.

3. Simplifying Complex Operations:

Certain problems can be solved more easily using specialized data structures. For example, if you're dealing with problems involving hierarchical data (like file systems or organizational charts), using tree data structures (e.g., binary trees, AVL trees) makes it easier to manage the data and perform operations like searching, adding, and deleting nodes.

4. Foundation for Algorithms:

Data structures are fundamental to many algorithms. Algorithms like sorting (quick sort, merge sort) and searching (binary search, depth-first search) rely heavily on the choice of data structure. A better data structure can lead to a more efficient algorithm, reducing computational costs.

5. Scalability and Performance:

As systems grow and handle more data, having an efficient data structure becomes critical for ensuring scalability. For example, databases often use **B-trees** or **hashing** techniques to efficiently index and retrieve large amounts of data.

6. Real-World Applications:

Data structures are widely used in real-world applications:

- **Databases** use tables (arrays) for indexing and searching records.
- **Networking** uses queues for managing packets of data.
- **Operating systems** rely on stacks and queues for process management.
- **Web browsers** use stacks (for managing the back-and-forward navigation) and trees (for the DOM structure).

7. Problem Solving and Algorithm Design:

The ability to choose and implement the appropriate data structure is essential for solving real-world problems. For example, in a graph traversal problem, you may choose a **depth-first search** (DFS) or **breadth-first search** (BFS) algorithm, and these algorithms rely on stacks and queues, respectively.

Conclusion

In conclusion, **data structures** are essential for efficiently organizing and manipulating data, and they play a central role in algorithms and system design. Understanding the **abstract data type** concept helps in focusing on the operations and functionality of data structures, while knowledge of their **importance** empowers developers to choose the right structure for the task, ensuring optimal performance. Data structures form the backbone of computer science and are indispensable in designing scalable and efficient systems.

The Stack

The Stack (3 hrs)

A **stack** is a linear data structure that follows the **Last In First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. The stack is often used in situations where we need to reverse data or manage processes that must be completed in reverse order. Stacks are used widely in algorithms, function calls, parsing expressions, and other computational processes.

---

1. Introduction to Stack

A **stack** is a collection of elements with two main operations:

- **Push**: Add an element to the stack.
- **Pop**: Remove the top element from the stack.

The stack follows the **LIFO** (Last In, First Out) order. This means that elements are added to and removed from the same end, known as the **top** of the stack. A real-world analogy for a stack is a pile of plates. When a new plate is added, it goes on top, and when a plate is removed, the one on top is taken away.

Stacks are used in many computer applications, such as:

- Function calls in recursion (each recursive call is pushed onto the stack).
- Undo operations in software (each previous state is pushed and popped from a stack).
- Expression evaluation (infix, postfix, and prefix).

---

2. Stack as an Abstract Data Type (ADT)

An **Abstract Data Type (ADT)** is a high-level description of a data structure that specifies its behavior, not the implementation details. A **stack** as an ADT can be defined as:

- **Operations**:
    - **Push(element)**: Insert an element onto the stack.
    - **Pop()**: Remove and return the top element of the stack.
    - **Peek()**: Return the top element without removing it.
    - **isEmpty()**: Check if the stack is empty.
    - **Size()**: Return the number of elements in the stack.

The **abstract** nature of the stack allows it to be implemented in various ways, such as using arrays, linked lists, or even dynamic resizing techniques, but its operations and behavior remain consistent across all implementations.

---

3. POP and PUSH Operations

The **PUSH** and **POP** operations are the fundamental operations that define the stack's behavior:

PUSH Operation

- **Purpose**: The PUSH operation adds an element to the stack.
- **How It Works**:
    - The new element is added to the top of the stack.
    - If implemented using an array, the top index is incremented, and the new element is placed at the new top.

- If implemented using a linked list, a new node is created and linked to the current top node.

Example:

stack = []

stack.append(10) # PUSH 10 to the stack

stack.append(20) # PUSH 20 to the stack

POP Operation

- **Purpose**: The POP operation removes and returns the top element of the stack.
- **How It Works**:
  - The top element is removed from the stack.
  - If implemented using an array, the top index is decremented.
  - If implemented using a linked list, the node at the top is removed, and the next node becomes the new top.

Example:

top = stack.pop() # POP the top element from the stack

Other Operations

- **Peek/Top**: Returns the element at the top of the stack without removing it.
- top_element = stack[-1] # Returns the top element without popping it
- **isEmpty**: Checks whether the stack is empty or not.
- if not stack: # Returns True if the stack is empty
- print("Stack is empty")

---

4. Stack Applications

Stacks have many practical applications, especially in computational problems involving recursion, expression evaluation, and backtracking. Some important applications of stacks include:

1. Evaluation of Infix, Postfix, and Prefix Expressions

Stacks are commonly used to evaluate expressions written in **infix**, **postfix**, and **prefix** notations. Let's look at how stacks are used to evaluate these expressions:

Infix Expression Evaluation

Infix notation is the common way of writing expressions, where operators are placed between operands (e.g., A + B). However, this notation requires parentheses to enforce precedence rules, making evaluation difficult without stacks.

To evaluate infix expressions using a stack:

1. Convert the infix expression to postfix notation using a stack.
2. Evaluate the postfix expression.

Postfix Expression Evaluation

In postfix notation (also known as Reverse Polish Notation), operators follow their operands. Postfix expressions are easier to evaluate because they do not require parentheses to denote operator precedence.

Steps for evaluating a postfix expression:

- Read the postfix expression from left to right.
- Push operands onto the stack.
- When an operator is encountered, pop operands from the stack, apply the operator, and push the result back onto the stack.
- After processing all symbols, the result will be at the top of the stack.

Example: Evaluate the postfix expression: 5 3 + 4 *

- Push 5 and 3 onto the stack.
- Encounter +, pop 3 and 5, calculate 5 + 3 = 8, and push the result 8.
- Push 4 onto the stack.
- Encounter *, pop 4 and 8, calculate 8 * 4 = 32, and push the result 32.
- The final result is 32.

Prefix Expression Evaluation

Prefix notation (also known as Polish Notation) places operators before operands. To evaluate a prefix expression:

1. Read the expression from right to left.
2. Push operands onto the stack.
3. When an operator is encountered, pop operands from the stack, apply the operator, and push the result back onto the stack.
4. The result will be at the top of the stack after processing all symbols.

---

2. Conversion of Expressions

Stacks are also used to convert expressions between infix, postfix, and prefix notations. The conversion process involves scanning the expression and applying the stack-based algorithm for each type of expression.

Infix to Postfix Conversion

To convert an infix expression to a postfix expression:

1. Scan the infix expression from left to right.
2. Operands (numbers or variables) are directly added to the output.
3. Operators are pushed onto the stack, but operators with higher precedence are popped first.
4. Parentheses are handled specially: open parentheses are pushed onto the stack, and when a closing parenthesis is encountered, operators are popped until an open parenthesis is found.

Example: Convert A + B * C to postfix.

- Read A, add to output: A
- Read +, push to stack: +
- Read B, add to output: A B
- Read *, push to stack: + *
- Read C, add to output: A B C
- Pop from the stack to get the final output: A B C * +

Result: **A B C * +**

Conclusion

In conclusion, the **stack** is a simple yet powerful data structure that plays a crucial role in algorithm design and problem-solving. Its **LIFO** nature makes it ideal for applications like expression evaluation, syntax parsing, backtracking problems, and recursive function calls. Understanding the operations of **push**, **pop**, and **peek**, along with its applications in **expression evaluation** and **conversion**, forms the foundation for using stacks effectively in programming.

Queue

Queue (3 hrs)

A **queue** is a linear data structure that follows the **First In First Out (FIFO)** principle, meaning the first element added to the queue is the first one to be removed. This principle makes it suitable for scenarios where data needs to be processed in the order it arrives, such as in scheduling tasks, handling requests,

or buffering data. In a queue, elements are inserted at one end (the rear) and removed from the other end (the front).

---

1. Introduction to Queue

A **queue** is a collection of elements with two primary operations:

- **Enqueue**: Insert an element at the end (rear) of the queue.
- **Dequeue**: Remove an element from the front of the queue.

Queues are used in scenarios where items need to be processed in a sequential manner, such as:

- **Print queue** in printers.
- **Task scheduling** in operating systems.
- **Packet buffering** in networks.
- **Breadth-first search (BFS)** in graph algorithms.

Queues are also used in various applications such as:

- **Handling requests** in web servers.
- **Managing tasks** in operating systems.
- **Simulating real-world scenarios** like customer service lines.

---

2. Queue as an Abstract Data Type (ADT)

A **Queue** is defined as an **Abstract Data Type (ADT)**, which describes the operations that can be performed on a queue, without specifying the details of how they are implemented. The queue as an ADT provides the following operations:

Basic Queue Operations:

1. **Enqueue(element)**: Adds an element to the back of the queue.
2. **Dequeue()**: Removes and returns the front element from the queue.
3. **Front()**: Returns the front element without removing it.
4. **isEmpty()**: Checks if the queue is empty.
5. **Size()**: Returns the number of elements currently in the queue.
6. **Clear()**: Removes all elements from the queue.

Queue Characteristics:

- **FIFO (First In, First Out)**: The first element enqueued is the first to be dequeued.

- The queue can be implemented using different underlying data structures such as arrays or linked lists.

---

3. Primitive Operations in Queue

Enqueue Operation

- **Purpose**: Adds an element to the back (rear) of the queue.
- **How it Works**:
    - The new element is placed at the rear of the queue.
    - If using an array, the rear index is incremented to add the element.
    - If using a linked list, a new node is created and attached to the rear.

Example:

```
queue = []

queue.append(10)  # Enqueue 10 to the queue

queue.append(20)  # Enqueue 20 to the queue
```

Dequeue Operation

- **Purpose**: Removes the element from the front of the queue and returns it.
- **How it Works**:
    - The element at the front is removed and returned.
    - If using an array, the front index is incremented to remove the element.
    - If using a linked list, the node at the front is removed and the next node becomes the new front.

Example:

```
front_element = queue.pop(0)  # Dequeue the front element
```

Front Operation

- **Purpose**: Retrieves the element at the front of the queue without removing it.
- **How it Works**:
    - Returns the element at the front of the queue.

Example:

```
front_element = queue[0]  # View the front element without removing it
```

isEmpty Operation

- **Purpose**: Checks if the queue is empty.
- **How it Works**:
    - Returns True if the queue is empty, otherwise False.

Example:

```
if not queue:  # Check if the queue is empty
   print("Queue is empty")
```

Size Operation

- **Purpose**: Returns the number of elements in the queue.
- **How it Works**:
    - Returns the count of elements currently in the queue.

Example:

```
queue_size = len(queue)  # Get the number of elements in the queue
```

---

4. Linear and Circular Queue and Their Applications

Linear Queue

In a **linear queue**, elements are arranged sequentially, and the first element is at the front, with the last element at the rear. However, when elements are dequeued, the space at the front becomes wasted, which can lead to inefficient memory utilization, even if there is space at the back.

- **Limitations**:
    - Once the rear reaches the end of the array, no new elements can be enqueued even if there is space at the front.

Example of Linear Queue:

```
queue = [1, 2, 3, 4]
queue.pop(0)  # Dequeue the first element
```

Circular Queue

In a **circular queue**, the queue is treated as circular, meaning the end of the queue wraps around to the beginning when space is available. This helps overcome the problem of wasted space in a linear queue, making it more efficient for use in situations where space is limited.

- **How it Works**:

- In a circular queue, when the rear reaches the end of the array, it “wraps around” to the beginning of the array (if there is space).
- The front and rear indices are updated in a modular fashion to allow the wraparound.

Example of Circular Queue:

queue = [None, None, None, None] # Size 4

front = 0

rear = -1

Applications of Linear and Circular Queues:

- **Linear Queue**: Common in situations where tasks need to be processed in the order they arrive (e.g., print spooling, handling HTTP requests).
- **Circular Queue**: Often used in applications like **buffering**, **round-robin scheduling**, or **networking protocols** (e.g., managing data packets in a buffer).

---

5. Priority Queue

A **priority queue** is a special type of queue where each element is assigned a **priority**. Elements with higher priority are dequeued before elements with lower priority, regardless of their order of insertion. In a regular queue, elements are dequeued in the order they are enqueued, but in a priority queue, the dequeue operation depends on the priority of the elements.

Priority Queue Operations:

- **Enqueue(element, priority)**: Insert an element into the queue with a specified priority.
- **Dequeue()**: Remove and return the element with the highest priority.

How It Works:

- Elements in the priority queue are sorted based on their priority, either in ascending or descending order.
- Internally, a priority queue can be implemented using a heap data structure (binary heap), where the root contains the highest priority element.

Applications of Priority Queue:

- **Task Scheduling**: Used in operating systems to schedule tasks with different priorities.
- **Dijkstra’s Algorithm**: Used in graph algorithms like Dijkstra’s shortest path algorithm to manage nodes with varying priorities.
- **Huffman Coding**: Used in data compression algorithms where characters are assigned priorities based on their frequency.

Example: A simple priority queue could be implemented by sorting elements by priority:

```
from queue import PriorityQueue


pq = PriorityQueue()
pq.put((1, "Task 1"))  # (priority, task)
pq.put((3, "Task 3"))
pq.put((2, "Task 2"))


while not pq.empty():
    priority, task = pq.get()
    print(f"Priority {priority}: {task}")
```

Priority Queue vs Regular Queue:

- **Regular Queue**: Follows **FIFO** (First In, First Out).
- **Priority Queue**: Serves elements based on **priority**, not the order of insertion.

Conclusion

The **queue** is a fundamental linear data structure that follows the **FIFO** principle. It provides two main operations: **enqueue** (to add an element) and **dequeue** (to remove an element), and is used in many real-world applications, such as task scheduling and buffering. **Linear queues** are simple but have limitations in space utilization, while **circular queues** efficiently address this issue. Additionally, a **priority queue** allows elements to be processed based on their priority, rather than their arrival order, making it useful for scheduling and algorithmic purposes.

List

List (2 hrs)

A **list** is a linear data structure that holds an ordered collection of elements. Unlike arrays, lists are flexible in terms of their size, which can grow or shrink dynamically. Lists are used widely in programming to store collections of data, such as sequences of items or objects, and provide functionality for easy access and manipulation.

---

1. Introduction to List

A **list** is a collection of elements, where each element is positioned in a specific order. The elements can be of any data type (integers, strings, objects, etc.), and the list allows insertion, deletion, and access of elements at any position. The key operations on lists are:

- **Insert**: Add an element to the list.
- **Delete**: Remove an element from the list.
- **Access**: Retrieve an element at a given position in the list.
- **Search**: Find the position of an element in the list.

Types of Lists:

1. **Singly Linked List**: A list where each element points to the next element.
2. **Doubly Linked List**: Each element points to both the next and the previous element.
3. **Array-based List**: A list implemented using a static array.

---

2. Static and Dynamic List Structure

Static List Structure (Array-based List)

In a **static list**, the size of the list is predefined and cannot be changed once it is created. It is implemented using an array, and elements are stored in contiguous memory locations. This makes accessing an element by index very fast, but resizing the list requires creating a new array.

Key Features:

- Fixed size (determined at the time of creation).
- Fast random access to elements via indices.
- Insertion and deletion of elements can be slow due to the need to shift elements in the array.

Example:

```
# Static list implemented using an array (Python List)

arr = [1, 2, 3, 4, 5]

arr[2] = 10  # Modify the element at index 2
```

Advantages:

- Quick access to elements using indices.
- Less memory overhead compared to dynamic lists.

Disadvantages:

- Fixed size (size must be defined upfront).

- Insertion and deletion are inefficient, as they may require shifting elements.

Dynamic List Structure

In a **dynamic list**, the size of the list can grow or shrink as elements are added or removed. This type of list is implemented using a dynamic data structure such as a linked list, where memory allocation can be adjusted during runtime.

Key Features:

- **Dynamic sizing**: The list can grow or shrink as needed.
- **Memory-efficient**: Memory is allocated as the list grows, and freed when elements are removed.
- **Increased flexibility**: Elements can be inserted or deleted without the need for shifting other elements.

Example:

```
# Dynamic list implemented using a linked list (Python List example)

dynamic_list = []

dynamic_list.append(1)  # Add element to the end of the list

dynamic_list.append(2)
```

Advantages:

- Flexible size.
- Efficient insertion and deletion of elements.

Disadvantages:

- Slower access time compared to arrays (because elements are not contiguous in memory).
- Additional memory overhead for storing pointers (in linked lists).

---

3. Array Implementation of Lists

In **array-based list implementation**, the list is implemented using an array (fixed-size or resizable). The list elements are stored in contiguous memory locations, allowing constant-time access to elements using an index.

Array-based Implementation Steps:

1. **Initialize the array**: Create an array with a fixed or dynamic size to store the list elements.
2. **Insert an element**: Insert elements at the end, or at a specified index by shifting elements if necessary.

3. **Delete an element**: Remove an element by shifting subsequent elements to fill the gap.
4. **Resize the array**: If the array reaches its capacity, create a new array with a larger size and copy elements to the new array.

Example:

```
# Array-based implementation (Python List)

array_list = [10, 20, 30, 40]

array_list.append(50)  # Add 50 at the end

array_list[2] = 35     # Update the element at index 2
```

Limitations:

- **Fixed size**: For static arrays, the size must be known ahead of time.
- **Resize overhead**: When the array is full and needs resizing, the time complexity can be high.

---

4. Queue as a List

A **queue** can be implemented as a list, where the elements are added at the rear (end) and removed from the front (beginning). This structure follows the **First In, First Out (FIFO)** principle, meaning that the first element added is the first to be removed.

In a queue implemented as a list:

- **Enqueue operation** adds an element at the end of the list.
- **Dequeue operation** removes an element from the front of the list.

Queue Operations in List:

1. **Enqueue**: Insert at the rear of the list.
2. **Dequeue**: Remove from the front of the list.
3. **Peek**: View the front element without removing it.
4. **isEmpty**: Check if the queue is empty.
5. **Size**: Get the number of elements in the queue.

Example:

```
# Implementing Queue using a List (Python)

queue = []

queue.append(10)  # Enqueue 10

queue.append(20)  # Enqueue 20
```

```
queue.pop(0)      # Dequeue (removes 10)
```

Efficiency Considerations:

- **Insertions (Enqueue)**: Adding an element to the rear of the list is efficient.
- **Deletions (Dequeue)**: Removing the front element can be inefficient if using a list because elements must be shifted.

To mitigate the inefficiency of **dequeue operations**, **circular queues** or **deque (double-ended queue)** can be used, which provide better performance.

Conclusion

A **list** is a versatile data structure used to store collections of data. It can be implemented in **static** or **dynamic** forms, with **arrays** providing fast access but limited size, and **linked lists** offering flexibility but slower access. A **queue** can be implemented using a list, providing FIFO behavior. Lists are fundamental structures in computing and serve as the foundation for many other data structures. Understanding the properties and operations of lists is crucial for designing efficient algorithms and systems.

Linked List

Linked Lists (5 hrs)

A **linked list** is a linear data structure consisting of a sequence of elements, where each element points to the next one in the sequence. Unlike arrays, linked lists are dynamic in size and can easily grow or shrink. Each element in a linked list is called a **node**, and each node typically contains two parts: the **data** and a reference (or **link**) to the next node in the list.

---

1. Introduction to Linked Lists

A **linked list** is a collection of nodes where each node contains two parts:

- **Data**: The value or the information stored in the node.
- **Link/Pointer**: A reference to the next node in the sequence.

Types of Linked Lists:

1. **Singly Linked List**: Each node points to the next node in the sequence.
2. **Doubly Linked List**: Each node points to both the next and the previous node.
3. **Circular Linked List**: The last node points back to the first node, making it circular.

Linked List Characteristics:

- **Dynamic size**: Unlike arrays, the size of the linked list is not fixed and can grow or shrink dynamically.
- **Sequential access**: Elements are accessed sequentially, starting from the head node.
- **Efficient insertions and deletions**: Insertions and deletions are easier in a linked list because only the links need to be updated, and no elements need to be shifted like in an array.

---

2. Linked List as an Abstract Data Type (ADT)

A **linked list** is considered an **Abstract Data Type (ADT)**, which means it defines the operations that can be performed on the linked list but does not specify the implementation details. The linked list ADT supports the following operations:

Linked List Operations:

1. **Insert**: Add a node to the linked list.
    - **At the beginning** (Head).
    - **At the end** (Tail).
    - **At a specific position**.
2. **Delete**: Remove a node from the linked list.
    - **From the beginning** (Head).
    - **From the end** (Tail).
    - **From a specific position**.
3. **Search**: Find a node based on its value.
4. **Traverse**: Visit each node in the linked list and perform an operation on the data.
5. **Size**: Get the number of elements in the list.
6. **IsEmpty**: Check if the linked list is empty.
7. **Clear**: Remove all nodes from the list.

---

3. Dynamic Implementation of Linked Lists

The primary advantage of linked lists is that they are **dynamically allocated**, meaning they can expand or contract in size as needed, without the need to predefine a size like arrays. This flexibility is possible because each node contains a reference to the next node, and memory for each node is allocated when it is created.

Memory Allocation:

- Each node is dynamically allocated in memory at runtime, which ensures efficient use of memory.
- Memory is freed when nodes are deleted, ensuring there is no memory waste.

Structure of a Singly Linked List:

A singly linked list typically has a **head** pointer that points to the first node. Each node has two components:

- **Data**: Stores the information.
- **Next**: Points to the next node in the list (or null if it is the last node).

Example (Singly Linked List):

```
class Node:
    def __init__(self, data=None):
        self.data = data  # Data stored in the node
        self.next = None  # Pointer to the next node


class LinkedList:
    def __init__(self):
        self.head = None  # Initialize the list with an empty head


    def insert_at_end(self, data):
        new_node = Node(data)  # Create a new node with data
        if not self.head:
            self.head = new_node  # If the list is empty, make the new node the head
        else:
            current = self.head
            while current.next:
                current = current.next  # Traverse to the end of the list
            current.next = new_node  # Add the new node at the end
```

---

4. Insertion and Deletion of Nodes

Insertion:

Insertion operations in a linked list are straightforward since we can easily adjust the pointers to insert a new node.

1. **Insert at the beginning**:
    - Create a new node.
    - Point the new node's next to the current head.
    - Update the head to point to the new node.
2. **Insert at the end**:
    - Traverse to the last node.
    - Point the last node's next to the new node.
3. **Insert at a specific position**:
    - Traverse the list to the desired position.
    - Point the previous node's next to the new node, and the new node's next to the next node.

Deletion:

Deleting a node requires adjusting the next pointers to remove a node from the list.

1. **Delete from the beginning**:
    - Point the head to the second node (i.e., head = head.next).
2. **Delete from the end**:
    - Traverse to the second-to-last node.
    - Set the second-to-last node's next to None.
3. **Delete from a specific position**:
    - Traverse to the node before the one to be deleted.
    - Point the previous node's next to the node after the one to be deleted.

Example (Insertion at the Beginning):

```
def insert_at_beginning(self, data):
    new_node = Node(data)  # Create a new node
    new_node.next = self.head  # Point new node to the current head
    self.head = new_node  # Update the head to point to the new node
```

---

5. Linked Stacks and Queues

A **stack** and a **queue** are abstract data types that can be efficiently implemented using linked lists.

Linked Stack:

A stack follows the **Last In, First Out (LIFO)** principle, where elements are added and removed from the same end (the top). In a **linked stack**:

- The **top** is the head node.
- **Push** operation inserts a new node at the head.
- **Pop** operation removes the node from the head.

Example (Linked Stack):

```
class Stack:
    def __init__(self):
        self.head = None

    def push(self, data):
        new_node = Node(data)  # Create new node
        new_node.next = self.head  # Point new node to the current head
        self.head = new_node  # Update the head to the new node

    def pop(self):
        if self.head:
            popped_data = self.head.data
            self.head = self.head.next  # Update head to the next node
            return popped_data
```

Linked Queue:

A queue follows the **First In, First Out (FIFO)** principle, where elements are added at the rear and removed from the front. In a **linked queue**:

- The **front** is the head node.
- The **rear** is the last node.

- **Enqueue** operation adds a node at the rear.
- **Dequeue** operation removes a node from the front.

Example (Linked Queue):

```
class Queue:
    def __init__(self):
        self.front = self.rear = None

    def enqueue(self, data):
        new_node = Node(data)
        if not self.rear:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def dequeue(self):
        if not self.front:
            return None
        dequeued_data = self.front.data
        self.front = self.front.next
        return dequeued_data
```

---

6. Doubly Linked Lists and Its Advantages

A **doubly linked list** is a variation of the singly linked list where each node contains two pointers: one pointing to the next node and another pointing to the previous node. This allows for more flexible traversal in both directions (forward and backward).

Structure of a Doubly Linked List:

- **Data**: Stores the data element.
- **Next**: Points to the next node in the list.

- **Prev**: Points to the previous node in the list.

Advantages of Doubly Linked Lists:

- **Bidirectional traversal**: Allows traversal in both directions, making it easier to navigate the list.
- **Efficient deletions**: Deleting a node can be done more efficiently, as both previous and next nodes are directly accessible.
- **Easier insertion and deletion at both ends**: Insertion and deletion operations at both the beginning and the end are simpler compared to a singly linked list.

Example (Doubly Linked List):

```
class DoublyNode:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_end(self, data):
        new_node = DoublyNode(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            new_node.prev = current
```

Conclusion

Linked lists are dynamic data structures that consist of nodes connected by pointers. They offer flexibility in terms of size and efficient insertion/deletion of nodes. **Singly linked lists** are simple but allow only unidirectional traversal, while **doubly linked lists** enable bidirectional traversal

Recursion

Recursion (4 hrs)

Recursion is a programming technique where a function calls itself in order to solve a problem. This method is widely used in problems where the solution can be broken down into smaller subproblems that have a similar structure to the original problem.

---

1. Introduction to Recursion

Recursion is a technique in computer science in which a function solves a problem by solving smaller instances of the same problem. The function continues to call itself until it reaches a base case, at which point it stops and begins returning the results. Recursive functions are defined in terms of themselves, and they are generally used for problems that can be divided into similar subproblems.

Basic Structure of Recursion:

A recursive function typically has two key components:

1. **Base Case**: The condition under which the function stops calling itself and returns a value. It ensures that the recursion does not continue indefinitely.
2. **Recursive Case**: The part of the function where it calls itself with a simpler version of the problem, gradually breaking the problem down until the base case is reached.

---

2. Principle of Recursion

The principle of recursion is based on the idea that a function can solve a problem by solving smaller instances of the same problem. It works through two phases:

- **Recursive Phase**: The function calls itself with a simpler or reduced version of the problem.
- **Base Case Phase**: When the function reaches a point where the problem is simple enough to be solved directly, thus terminating the recursion.

Example (Factorial Calculation):

The factorial of a number n is the product of all positive integers less than or equal to n:

$n!=n\times(n-1)\times(n-2)\times...\times1$

The factorial can be defined recursively as:

$n! = n \times (n-1)!$

And the base case would be:

$0! = 1$

Recursive Function:

```
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n-1)  # Recursive case
```

---

3. Recursion vs. Iteration

Recursion:

- Recursion involves a function calling itself until it reaches the base case.
- Recursion can lead to simpler, more readable code for problems that naturally fit a recursive structure, such as tree traversal or computing factorials.
- However, recursion can be less efficient in terms of memory usage and performance, as each function call adds a new layer to the call stack.

Iteration:

- Iteration uses loops to repeat a set of instructions until a condition is met.
- It is often more efficient than recursion, particularly in terms of memory usage, as it does not require maintaining a call stack.
- Iteration may require more complex logic when dealing with problems that have a recursive structure.

Comparison:

Aspect	Recursion	Iteration
Memory Usage	Uses stack for each function call	Uses loop variable(s) only

Aspect	Recursion	Iteration
Code Simplicity	Simple code for recursive problems	More complex for recursive problems
Efficiency	Can be less efficient (stack overhead)	Often more efficient
Use Case	Ideal for tree-based and divide-and-conquer problems	Ideal for simple repetitive tasks

---

4. Recursion Examples

Tower of Hanoi (TOH)

The **Tower of Hanoi** problem involves moving a stack of disks from one rod to another, obeying three rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. Only the topmost disk of a stack can be moved.

Recursive Solution to Tower of Hanoi:

The problem can be solved recursively by breaking it down into three steps:

1. Move n−1 disks from the source rod to the auxiliary rod.
2. Move the largest disk from the source rod to the destination rod.
3. Move the n−1 disks from the auxiliary rod to the destination rod.

```
def tower_of_hanoi(n, source, auxiliary, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
    else:
        tower_of_hanoi(n-1, source, destination, auxiliary)
        print(f"Move disk {n} from {source} to {destination}")
        tower_of_hanoi(n-1, auxiliary, source, destination)
```

```
# Example usage:
tower_of_hanoi(3, 'A', 'B', 'C')
```

Fibonacci Series

The Fibonacci sequence is defined as:

$F(n)=F(n-1)+F(n-2)$

With base cases:

$F(0)=0, F(1)=1$

Recursive Solution for Fibonacci:

```
def fibonacci(n):
    if n == 0:
        return 0  # Base case
    elif n == 1:
        return 1  # Base case
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # Recursive case


# Example usage:
print(fibonacci(5))  # Output: 5
```

---

5. Applications of Recursion

Recursion is widely used in various fields of computer science, especially in problems where a task can be broken down into smaller subproblems that are similar to the original problem.

Common Applications:

1. **Tree Traversal**: Recursion is ideal for traversing trees (binary trees, etc.). Examples include in-order, pre-order, and post-order traversal.
2. **Divide and Conquer Algorithms**: Problems like **Merge Sort** and **Quick Sort** use recursion to divide the problem into smaller subproblems and combine the results.
3. **Graph Traversal**: Recursion is used in graph traversal algorithms like **Depth-First Search (DFS)**.
4. **Backtracking Algorithms**: Problems like **N-Queens**, **Sudoku Solver**, and **Maze Solving** often use recursion.

5. **Dynamic Programming**: Recursion is used in problems that require breaking down a problem into overlapping subproblems, such as in **Fibonacci sequence calculation** or **Knapsack problem**.

---

6. Search Tree

A **search tree** is a tree data structure in which each node stores a value greater than all values in its left subtree and smaller than all values in its right subtree. Searching in a search tree can be efficiently performed using recursion.

Recursive Search in a Binary Search Tree (BST):

To search for a value in a binary search tree:

1. If the tree is empty, return None.
2. If the value is equal to the root, return the root.
3. If the value is smaller than the root, recursively search the left subtree.
4. If the value is greater than the root, recursively search the right subtree.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def search(root, value):
    if root is None or root.value == value:
        return root
    if value < root.value:
        return search(root.left, value)
    return search(root.right, value)
```

---

7. Conclusion

Recursion is a powerful and elegant tool for solving problems that can be broken down into smaller subproblems. It is particularly useful for tasks involving hierarchical structures like trees and graphs, as well as for problems that can be solved through a divide-and-conquer approach. While recursion can

lead to clean, easy-to-understand solutions, it is important to understand when it is appropriate to use recursion over iteration, as it can be less efficient in terms of memory usage and performance.

Trees

Trees (5 hrs)

A **tree** is a hierarchical data structure used in computer science to represent relationships among elements in a way that mimics natural hierarchies. Trees consist of nodes connected by edges, and they are widely used for searching, sorting, and representing hierarchical data, such as organizational structures, file systems, and more.

---

1. Introduction to Trees

In computer science, a **tree** is a collection of nodes where each node contains a value and has references (or pointers) to its child nodes. A tree structure is used for various applications such as representing hierarchical structures, expression parsing, database indexing, and more.

Basic Tree Terminology:

- **Root**: The top node of the tree, which does not have any parent.
- **Node**: A fundamental unit that stores data and links to child nodes.
- **Edge**: A connection between two nodes.
- **Leaf**: A node with no children.
- **Subtree**: A tree formed by a node and all of its descendants.
- **Parent**: A node that has child nodes.
- **Child**: A node directly connected to another node when moving away from the root.
- **Sibling**: Nodes that share the same parent.

Types of Trees:

- **Binary Tree**: A tree where each node has at most two children (left and right).
- **Binary Search Tree (BST)**: A binary tree where the left child is smaller than the parent node, and the right child is larger.
- **AVL Tree**: A self-balancing binary search tree where the height of the two child subtrees of every node differs by no more than one.
- **B-Tree**: A self-balancing search tree designed for systems that read and write large blocks of data.

---

2. Basic Operations in Binary Tree

A **binary tree** is a tree where each node has at most two children, typically referred to as the left child and the right child. Basic operations performed on binary trees include:

Insertion:

- Inserting a node into a binary tree generally involves finding the appropriate place for the new node. In a binary search tree, for example, the node is inserted in the correct position based on its value.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def insert(root, value):
    if root is None:
        return TreeNode(value)
    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)
    return root
```

Search:

- Searching for a node in a binary search tree involves comparing the value to be searched with the root node. Based on the comparison, we either move left or right recursively.

```
def search(root, value):
    if root is None or root.value == value:
        return root
    if value < root.value:
        return search(root.left, value)
    return search(root.right, value)
```

Deletion:

- Deletion in a binary tree can be complex. If the node to be deleted has:
    - **No children**: Simply remove the node.
    - **One child**: Replace the node with its child.
    - **Two children**: Replace the node with its in-order successor (or predecessor).

---

3. Tree Search and Insertion/Deletion

Tree Search:

- **Search** is the process of finding a specific value in the tree.
- In a **binary search tree**, the search follows the property that all nodes on the left subtree are smaller and all nodes on the right subtree are larger than the root node. This allows for efficient search operations.

Insertion and Deletion:

- **Insertion** and **Deletion** in a binary search tree are recursive operations that maintain the tree's structure. Insertion involves placing the new value in the correct position, while deletion involves re-arranging the tree to maintain its properties.

---

4. Binary Tree Traversals

Tree traversal is the process of visiting all the nodes in a tree and performing some operation on each. There are three common types of traversals in a binary tree:

1. Pre-order Traversal:

In pre-order traversal, the root node is processed first, followed by the left subtree, and then the right subtree.

```
def pre_order_traversal(root):
    if root:
        print(root.value, end=' ')
        pre_order_traversal(root.left)
        pre_order_traversal(root.right)
```

2. In-order Traversal:

In in-order traversal, the left subtree is processed first, followed by the root node, and then the right subtree. This traversal is particularly useful in binary search trees, as it processes the nodes in ascending order.

```
def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.value, end=' ')
        in_order_traversal(root.right)
```

3. Post-order Traversal:

In post-order traversal, the left subtree is processed first, followed by the right subtree, and then the root node. This is useful for tasks like deletion, where you need to delete the children before the parent.

```
def post_order_traversal(root):
    if root:
        post_order_traversal(root.left)
        post_order_traversal(root.right)
        print(root.value, end=' ')
```

---

5. Tree Height, Level, and Depth

- **Height of a Tree**: The height of a tree is the length of the longest path from the root to any leaf. The height of a tree with no nodes is considered to be -1, and the height of a tree with only one node is 0.
- **Level of a Node**: The level of a node is the number of edges from the root to the node.
- **Depth of a Node**: The depth of a node is the same as the level of the node. It represents the distance from the root to the node.

```
def height(root):
    if root is None:
        return -1
    left_height = height(root.left)
    right_height = height(root.right)
    return max(left_height, right_height) + 1
```

---

6. Balanced Trees: AVL Trees

A **balanced tree** is a tree structure in which the height of the left and right subtrees of any node differ by at most one. This ensures that the tree remains balanced, leading to more efficient operations (search, insertion, and deletion).

AVL Trees:

An **AVL tree** (named after its inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. It ensures that the height difference between the left and right subtrees of any node is no more than one. If this balance property is violated, rotations are used to restore balance.

Balancing Algorithm:

- **Left Rotation**: Used when the right subtree is taller than the left.
- **Right Rotation**: Used when the left subtree is taller than the right.
- **Left-Right Rotation**: A combination of left and right rotations used when there is a left-heavy subtree on the right side.
- **Right-Left Rotation**: A combination of right and left rotations used when there is a right-heavy subtree on the left side.

```
# Simple example of rotations in AVL trees (not a complete implementation)

def left_rotate(x):
    y = x.right
    x.right = y.left
    y.left = x
    return y
```

---

7. Huffman Algorithm

The **Huffman coding algorithm** is used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. The algorithm uses a binary tree, known as a **Huffman tree**, to assign these codes.

Steps:

1. Build a frequency table for the characters.
2. Construct a priority queue (min-heap) where the lowest frequencies are given the highest priority.
3. Build the Huffman tree by repeatedly merging the two smallest trees in the queue.
4. Assign binary codes based on the tree structure.

---

8. Game Trees

A **game tree** is a tree representation of possible moves in a game. The nodes represent game states, and the edges represent player moves. Game trees are widely used in algorithms for games like chess, checkers, and tic-tac-toe.

Minimax Algorithm:

The **minimax algorithm** is used to minimize the possible loss for a worst-case scenario. It is a recursive algorithm for choosing the optimal move in a two-player game. The algorithm simulates all possible moves and selects the best one.

---

9. B-Trees

A **B-tree** is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. B-trees are used in databases and file systems due to their efficiency in handling large amounts of data.

B-tree Properties:

- Each node can have multiple children (more than two).
- Nodes store multiple keys, which help in narrowing down searches.
- All leaf nodes are at the same level, ensuring balanced growth.

Conclusion

Trees are a fundamental data structure that plays a crucial role in organizing and accessing data efficiently. Operations like search, insertion, and deletion are optimized in tree structures such as binary trees and B-trees. Advanced topics like AVL trees, Huffman coding, and game trees provide practical solutions for more complex computational problems, ensuring faster processing, better data management, and improved performance across a wide range of applications.

Sorting

Sorting (5 hrs)

Sorting is a fundamental operation in computer science and programming, used to arrange elements in a specific order (either ascending or descending). Sorting plays an essential role in many applications, such as searching, indexing, data processing, and optimizing algorithms.

---

1. Introduction to Sorting

Sorting is the process of rearranging a sequence of elements in a particular order. The order can be numerical, alphabetical, or based on custom criteria. Sorting is crucial for efficient data retrieval, optimization, and simplification of problems.

Types of Sorting Algorithms:

- **Internal Sorting**: The data to be sorted fits entirely into the computer's memory.
- **External Sorting**: The data is too large to fit into memory and must be stored on external storage (e.g., disk).

---

2. Internal and External Sorting

Internal Sorting:

- This type of sorting occurs when the entire dataset is small enough to be held in the computer's main memory (RAM).
- Algorithms used for internal sorting include Quick Sort, Merge Sort, Insertion Sort, Selection Sort, and Heap Sort.

External Sorting:

- External sorting is necessary when the dataset is too large to fit into memory. Data is stored in external storage like hard drives, and a portion of data is loaded into memory, sorted, and then written back to disk.
- A typical use case of external sorting is when dealing with large databases and files that cannot be processed in memory.
- A popular algorithm for external sorting is the **Merge Sort** as it works efficiently with external data by using a "divide and conquer" approach.

---

3. Insertion Sort

Insertion Sort:

- **Concept**: Insertion sort works similarly to the way you sort playing cards. It picks an element and places it in the correct position among the already sorted elements.
- **Procedure**: The algorithm starts with the second element, compares it with the first, and inserts it in the correct position. It continues for each subsequent element until the entire array is sorted.

Time Complexity:

- Best Case: O(n) (when the array is already sorted).
- Worst Case: O(n2) (when the array is in reverse order).

- Average Case: O(n2).

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

---

4. Selection Sort

Selection Sort:

- **Concept**: Selection sort works by repeatedly finding the minimum element from the unsorted portion and swapping it with the first unsorted element.
- **Procedure**: In each pass, it selects the smallest (or largest) element from the unsorted portion and moves it to the sorted portion.

Time Complexity:

- Best, Worst, and Average Case: O(n2) as it always performs the same number of comparisons regardless of the initial order of elements.

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

---

5. Exchange Sort (Bubble Sort)

Bubble Sort:

- **Concept**: Bubble sort works by comparing adjacent elements and swapping them if they are in the wrong order. The process is repeated until no swaps are needed.
- **Procedure**: This sorting algorithm “bubbles” the largest element to the top in each pass, hence the name.

Time Complexity:

- Best Case: O(n) (when the list is already sorted).
- Worst and Average Case: O(n2).

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
```

---

6. Quick Sort

Quick Sort:

- **Concept**: Quick Sort is a “divide and conquer” algorithm that works by selecting a pivot element, partitioning the array around the pivot, and recursively sorting the subarrays.
- **Procedure**:
    - Choose a pivot element.
    - Partition the array into two subarrays: one with elements smaller than the pivot, and the other with elements greater than the pivot.
    - Recursively apply the quick sort to the subarrays.

Time Complexity:

- Best and Average Case: O(nlogn)
- Worst Case: O(n2) (when the pivot is the smallest or largest element in each partition).

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

---

7. Merge Sort

Merge Sort:

- **Concept**: Merge Sort is another “divide and conquer” algorithm that divides the array into two halves, sorts them, and then merges the sorted halves.
- **Procedure**:
    - Recursively divide the array into two halves.
    - Merge the sorted halves back together.

Time Complexity:

- Best, Worst, and Average Case: O(nlogn)

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0
```

```
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

---

8. Radix Sort

Radix Sort:

- **Concept**: Radix Sort is a non-comparative sorting algorithm that processes numbers digit by digit, from the least significant digit to the most significant.
- **Procedure**:
    - Sort the elements based on each digit starting from the least significant digit using a stable sorting algorithm (like Counting Sort).

Time Complexity:

- Best, Worst, and Average Case: O(nk), where n is the number of elements and k is the number of digits.

---

9. Shell Sort

Shell Sort:

- **Concept**: Shell Sort is an improvement on Insertion Sort that allows the exchange of items that are far apart.
- **Procedure**: It starts by sorting pairs of elements far apart and gradually reducing the gap between the elements to be compared.

Time Complexity:

- Best Case: O(nlogn) (for optimal gap sequence).

- Worst Case: O(n2) (for poor gap sequences).

---

10. Binary Sort

Binary Sort:

- **Concept**: Binary Sort is a variation of the Insertion Sort where binary search is used to find the correct position to insert each element, which reduces the number of comparisons.
- **Procedure**:
    - Use binary search to find the position of the element in the sorted portion and insert it accordingly.

---

11. Heap Sort (as a Priority Queue)

Heap Sort:

- **Concept**: Heap Sort is based on the **heap data structure** (usually a binary heap) that represents a complete binary tree. Heap Sort can be used to sort data by treating it as a priority queue.
- **Procedure**:
    - Convert the array into a max-heap.
    - Swap the root (maximum element) with the last element.
    - Heapify the root, reducing the heap size by 1, and repeat until the heap is empty.

Time Complexity:

- Best, Worst, and Average Case: O(nlogn)

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
```

```
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)

def heap_sort(arr):
  n = len(arr)
  for i in range(n//2 - 1, -1, -1):
    heapify(arr, n, i)
  for i in range(n-1, 0, -1):
    arr[i], arr[0] = arr[0], arr[i]
    heapify(arr, i, 0)
```

---

12. Efficiency of Sorting

The efficiency of sorting algorithms is commonly measured by their **time complexity** and **space complexity**. Time complexity is the number of comparisons or operations needed to sort the data, while space complexity is the amount of memory used by the algorithm.

Big O Notation:

- **Big O notation** is used to describe the upper bound of an algorithm's runtime in terms of the size of the input data.
- For sorting algorithms:
  - **O(n^2)**: Selection Sort, Insertion Sort, Bubble Sort.
  - **O(n log n)**: Merge Sort, Quick Sort, Heap Sort, Shell Sort.
  - **O(nk)**: Radix Sort (where k is the number of digits).

Conclusion

Sorting is a core component of many computational tasks, and understanding different sorting algorithms helps in choosing the right one for a given situation. Algorithms like Quick Sort, Merge Sort, and Heap Sort offer efficient solutions, while simpler ones like Insertion Sort and Selection Sort are useful for small datasets or when simplicity is preferred. Optimizing sorting algorithms for specific use cases can greatly improve performance.

Searching

Searching (5 hrs)

Searching is a fundamental concept in computer science and algorithms, aimed at finding specific data within a structure or database. A wide variety of searching techniques exist, each suited for different types of data structures and use cases. The goal of a searching algorithm is to efficiently locate an element in a dataset or verify whether the element is present or not.

---

1. Introduction to Search Techniques

Searching is the process of finding a specific item within a collection of data. Different types of search algorithms are used depending on the data structure being searched and the type of data. In some cases, searching can be done sequentially, while in other cases, more efficient algorithms like binary search or hashing are employed.

Essentials of Searching:

- **Search Key**: The element you're searching for.
- **Search Space**: The set of elements in which you're searching.
- **Search Outcome**: Whether the element is found or not. Typically, the outcome involves returning the index of the element or an indication that the element does not exist in the collection.

Searching algorithms vary in efficiency, depending on the structure of the data and the size of the dataset.

---

2. Sequential Search (Linear Search)

Concept:

- **Sequential Search** is the simplest searching technique where the algorithm scans each element in the collection, one by one, until the desired element is found or the entire list is searched.
- **Procedure**: Start from the first element, compare it with the search key, move to the next element, and repeat until either the element is found or the end of the list is reached.

Time Complexity:

- **Worst Case**: O(n), when the element is at the end of the list or not present at all.
- **Best Case**: O(1), if the element is at the beginning of the list.
- **Average Case**: O(n).

```
def sequential_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
```

```
            return index  # Return index if found
    return -1  # Return -1 if not found
```

---

3. Binary Search

Concept:

- **Binary Search** is an efficient searching algorithm that works on sorted data structures. It repeatedly divides the search space in half until the target element is found.
- **Procedure**:
    1. Begin by comparing the target with the middle element of the array.
    2. If the target is equal to the middle element, return its index.
    3. If the target is smaller than the middle element, continue the search in the left half.
    4. If the target is larger, continue the search in the right half.

Precondition:

- The array or list must be sorted in ascending or descending order before applying binary search.

Time Complexity:

- **Worst Case**: O(logn), as the search space is halved in each iteration.
- **Best Case**: O(1), if the middle element is the target.
- **Average Case**: O(logn).

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half
    return -1  # Target not found
```

---

4. Tree Search

Concept:

- **Tree Search** involves searching through a tree data structure, where each node contains a value and references to child nodes.
- **Binary Search Tree (BST)** is a common tree used for efficient searching. In a BST, for any node:
    - The left child contains smaller values than the node.
    - The right child contains larger values than the node.

Time Complexity:

- **Best Case**: O(logn), for a balanced tree.
- **Worst Case**: O(n), for a skewed tree (essentially a linked list).

Procedure:

1. Start from the root of the tree.
2. Compare the target with the current node.
3. Recursively search the left or right subtree depending on whether the target is smaller or larger than the current node.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def tree_search(root, target):
    if root is None or root.value == target:
        return root
    if target < root.value:
        return tree_search(root.left, target)
    return tree_search(root.right, target)
```

---

5. General Search Tree

Concept:

- **General Search Tree** (GST) is a type of tree where each node can have an arbitrary number of children, unlike Binary Search Trees (BSTs) which only have two children (left and right).
- **Application**: General search trees are used in scenarios where the data has a hierarchical structure with more complex relationships than just two branches.

Efficiency:

- Searching in a General Search Tree may require exploring multiple branches, which can affect the efficiency. Therefore, optimizing the tree structure and balancing the tree is important for improving performance.

---

6. Hashing: Hash Function and Hash Tables

Hashing:

- **Hashing** is a technique used to efficiently store and search data by mapping keys to specific locations in an array (called a hash table) using a hash function.
- **Hash Function**: A hash function takes an input (or "key") and returns an integer value, which is used as an index to store the data in a hash table.

Hash Table:

- A **hash table** is an array-like structure used to store data, where each key is mapped to a specific index in the array using the hash function.

Collision Resolution:

- **Collision** occurs when two keys hash to the same index. There are different techniques to handle collisions:
    1. **Chaining**: Store multiple elements at the same index using a linked list.
    2. **Open Addressing**: Find another open slot in the hash table using methods such as linear probing, quadratic probing, or double hashing.

Time Complexity:

- **Average Case**: O(1) for both searching and insertion if collisions are minimized.
- **Worst Case**: O(n) if many collisions occur and the hash table becomes overloaded.

```
class HashTable:
    def __init__(self, size):
        self.table = [None] * size
```

```
    def hash_function(self, key):
        return key % len(self.table)

    def insert(self, key, value):
        index = self.hash_function(key)
        if self.table[index] is None:
            self.table[index] = [(key, value)]
        else:
            self.table[index].append((key, value))

    def search(self, key):
        index = self.hash_function(key)
        if self.table[index] is not None:
            for k, v in self.table[index]:
                if k == key:
                    return v
        return None
```

---

7. Efficiency Comparisons of Different Search Techniques

1. Sequential Search:

- **Best Case**: O(1)
- **Worst Case**: O(n)
- **Average Case**: O(n)
- **Efficiency**: Very inefficient for large datasets, but simple and works on unsorted data.

2. Binary Search:

- **Best Case**: O(1)
- **Worst Case**: O(logn)
- **Average Case**: O(logn)

- **Efficiency**: Very efficient for sorted data. Much faster than sequential search for large datasets.

3. Tree Search (Binary Search Tree):

- **Best Case**: O(logn)
- **Worst Case**: O(n) (for skewed tree)
- **Average Case**: O(logn)
- **Efficiency**: Efficient if the tree is balanced. However, can degrade if the tree becomes unbalanced.

4. Hashing:

- **Best Case**: O(1)
- **Worst Case**: O(n) (if all keys hash to the same index)
- **Average Case**: O(1)
- **Efficiency**: Very efficient for large datasets with good hash functions and collision resolution techniques.

Conclusion

Searching is a crucial aspect of algorithmic problem-solving. Depending on the structure of the data and the requirements of the problem, different search techniques offer varying trade-offs between time complexity and space complexity. **Binary search** is highly efficient for sorted arrays, while **hashing** can provide constant time lookups for key-value pairs. **Tree-based searches** work well for hierarchical data but need balanced structures for optimal performance. Understanding these techniques helps in selecting the right one for any given situation, ensuring efficient data retrieval and system performance.

Graphs

Graphs (5 hrs)

Graphs are fundamental data structures used to model relationships or connections between objects. They are used extensively in various fields such as computer science, social networks, transportation networks, and many others. Graphs provide a versatile way to represent data and solve problems related to connectivity, shortest paths, and network flow.

---

1. Introduction to Graphs

A **graph** is a collection of **vertices** (also called nodes) and **edges** (connections or links between the nodes). Each edge connects two vertices and can either be directed or undirected.

Key Terminology:

- **Vertex (Node)**: A fundamental unit of a graph, representing an entity.
- **Edge**: A connection between two vertices, representing the relationship between them.
- **Adjacent**: Two vertices are adjacent if they are connected by an edge.
- **Degree**: The number of edges incident to a vertex. In an undirected graph, it's the number of edges connected to a vertex. In a directed graph, it can be divided into **in-degree** (edges coming into the vertex) and **out-degree** (edges going out of the vertex).
- **Path**: A sequence of vertices connected by edges.
- **Cycle**: A path where the first and last vertices are the same, and no other vertex is repeated.

Graphs can be classified into different types based on the edges and their properties.

---

2. Graphs as an Abstract Data Type (ADT)

A **Graph ADT** defines a set of operations that can be performed on a graph. These operations typically include:

1. **Adding a vertex**: Insert a new node into the graph.
2. **Adding an edge**: Create a connection between two vertices.
3. **Removing a vertex**: Delete a node from the graph.
4. **Removing an edge**: Delete a connection between two vertices.
5. **Checking adjacency**: Determine if two vertices are connected by an edge.
6. **Traversing**: Explore or visit all vertices and edges of the graph in some systematic way.

Graph implementations can be done using:

- **Adjacency Matrix**: A 2D array where the element at row i and column j is 1 if there is an edge between vertex i and vertex j.
- **Adjacency List**: A list where each vertex stores a list of vertices adjacent to it.

---

3. Transitive Closure

The **Transitive Closure** of a graph is a way of representing reachability in the graph. It tells you whether a path exists between two vertices, either directly or indirectly.

In a directed graph, the transitive closure of the graph is a matrix where each element (i, j) indicates whether there is a path from vertex i to vertex j.

Application:

- **Path Finding**: Used to determine if two nodes are connected, regardless of the path length.

- **Warshall's Algorithm**: An algorithm used to compute the transitive closure of a directed graph.

---

4. Warshall's Algorithm

Warshall's Algorithm is an efficient algorithm for finding the transitive closure of a directed graph. It works by updating the adjacency matrix to reflect reachability between all pairs of vertices.

Procedure:

1. Start with the adjacency matrix of the graph.
2. For each intermediate vertex k, update the matrix by checking if a path exists from vertex i to vertex j through vertex k.
3. Repeat the process for all intermediate vertices.

Time Complexity: O(V3), where V is the number of vertices in the graph.

---

5. Types of Graphs

Graphs can be classified based on various properties:

- **Directed Graph (Digraph)**: The edges have a direction, i.e., an edge from vertex u to vertex v is not the same as an edge from v to u.
- **Undirected Graph**: The edges do not have a direction, meaning if there is an edge between u and v, it is the same as an edge between v and u.
- **Weighted Graph**: The edges carry weights or costs that represent the strength or length of the connection between the vertices.
- **Unweighted Graph**: The edges do not carry any weights.
- **Connected Graph**: A graph is connected if there is a path between every pair of vertices.
- **Disconnected Graph**: A graph that is not connected, meaning there exist pairs of vertices with no path between them.
- **Cyclic Graph**: A graph that contains at least one cycle.
- **Acyclic Graph**: A graph with no cycles.

---

6. Graph Traversal and Spanning Forests

Graph traversal refers to the process of visiting all the vertices and edges of a graph in a systematic way. The two most common traversal algorithms are:

1. **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking.

- **Time Complexity**: O(V+E), where V is the number of vertices and E is the number of edges.

2. **Breadth-First Search (BFS)**: Explores all neighbors of a vertex before moving on to the next level of neighbors.
    - **Time Complexity**: O(V+E).

A **Spanning Tree** is a subgraph of a graph that includes all the vertices but only enough edges to make the graph connected (i.e., no cycles). A **Spanning Forest** is a collection of spanning trees for a disconnected graph.

---

7. Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a graph. The MST is a subgraph that connects all vertices with the minimum total edge weight, and no cycles are formed.

Procedure:

1. Sort all edges of the graph by their weight.
2. Select the edge with the smallest weight that does not form a cycle with the selected edges.
3. Repeat until the MST contains exactly V−1 edges (where V is the number of vertices).

Time Complexity: O(ElogE), where E is the number of edges in the graph.

---

8. Round-Robin Algorithm

Round-Robin Scheduling is a simple and widely used algorithm in scheduling systems, but it is also applicable in graph-related problems such as load balancing. It works by iterating through a set of tasks and distributing them equally in a cyclic manner.

In the context of graphs, a round-robin algorithm could be used to evenly distribute edges or weights across multiple nodes or components.

---

9. Shortest-Path Algorithm

The **Shortest Path Problem** is concerned with finding the shortest path from one vertex to another in a graph. Several algorithms exist for solving this problem, including **Dijkstra's Algorithm** and **Bellman-Ford Algorithm**.

Greedy Algorithm:

- A greedy algorithm makes the locally optimal choice at each stage, with the hope of finding the global optimum. Dijkstra's algorithm is a classic example of a greedy approach for finding the shortest path in weighted graphs.

---

10. Dijkstra's Algorithm

Dijkstra's Algorithm is a popular algorithm for finding the shortest paths from a source vertex to all other vertices in a graph. It works on **weighted graphs** (with non-negative edge weights) and is a greedy algorithm.

Procedure:

1. Initialize distances from the source vertex to all other vertices as infinity, except the source vertex, which has a distance of zero.
2. Mark the source vertex as visited and move to the nearest unvisited vertex with the smallest known distance.
3. Update the shortest distances for each adjacent vertex.
4. Repeat the process until all vertices are visited.

Time Complexity:

- With a simple array: O(V2)
- With a priority queue (binary heap): O((V+E)logV)

Conclusion

Graphs are versatile data structures used to represent a wide range of real-world problems, such as networking, social media, and logistics. The choice of graph traversal and algorithms such as Kruskal's, Dijkstra's, or Warshall's depends on the specific requirements of the problem, such as minimizing the path length, building spanning trees, or computing reachability. By understanding the various graph-related algorithms and their complexities, you can apply the appropriate technique to solve graph-based problems efficiently.

Algorithms

Algorithms (5 hrs)

An algorithm is a well-defined sequence of steps or rules designed to solve a problem or perform a task. Algorithms are the foundation of computer science and are used to process data, perform calculations, automate reasoning, and more. This topic explores the classification of algorithms into various types, including deterministic and non-deterministic algorithms, divide and conquer, series and parallel algorithms, as well as heuristic and approximate algorithms.

---

1. Deterministic and Non-Deterministic Algorithms

Deterministic Algorithm

A **deterministic algorithm** is an algorithm that, given a particular input, always produces the same output and follows a specific sequence of steps. The behavior of the algorithm is predictable, and there is no uncertainty or randomness in its execution. Each step is predefined, and there is no ambiguity in the execution flow.

Examples:

- **Binary Search**: Given a sorted array, it always returns the same result for the same input, and the search steps are well-defined.
- **Euclidean Algorithm**: This algorithm calculates the greatest common divisor (GCD) of two numbers in a deterministic manner.

Key Characteristics:

- Predictable behavior
- No randomness or ambiguity
- Same output for the same input

Non-Deterministic Algorithm

A **non-deterministic algorithm** is an algorithm where the execution can vary for the same input. The sequence of steps can involve decisions made randomly, or the algorithm might have multiple possible outcomes for a given input. In a non-deterministic algorithm, there is no guarantee that it will always follow the same path or yield the same result.

Examples:

- **Non-deterministic Turing Machine**: This type of theoretical machine can take multiple paths for a given input, and its behavior is not fixed.
- **Monte Carlo Algorithm**: These algorithms use random numbers to produce approximate solutions to problems. For the same input, it may generate different results each time.

Key Characteristics:

- Unpredictable behavior
- May involve random or probabilistic decisions
- Can produce different outputs for the same input

---

2. Divide and Conquer Algorithm

The **Divide and Conquer** technique is a fundamental algorithm design paradigm that involves breaking a problem down into smaller subproblems, solving each subproblem independently, and combining the results to obtain the solution to the original problem. It is highly effective for problems that have overlapping subproblems.

Steps:

1. **Divide**: Break the problem into smaller subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge the solutions of the subproblems to form the final solution.

Examples:

- **Merge Sort**: This sorting algorithm divides the array into smaller subarrays, recursively sorts them, and then merges them to get the sorted array.
- **Quick Sort**: Another sorting algorithm that divides the array based on a pivot and recursively sorts the subarrays.
- **Binary Search**: Involves dividing the search space into halves until the target is found.

Advantages:

- Efficient for large problems
- Reduces time complexity by solving smaller subproblems
- Often leads to parallelizable solutions

Time Complexity: In many cases, divide and conquer algorithms have a time complexity of O(nlogn), such as in Merge Sort and Quick Sort.

---

3. Series and Parallel Algorithms

Series Algorithm

A **series algorithm** is one where tasks are executed sequentially, i.e., one after another. Each step of the algorithm depends on the result of the previous step, and there is no parallelism involved. These algorithms are suitable for problems where the next step cannot be performed until the current step is completed.

Examples:

- **Bubble Sort**: The algorithm iterates through the array, compares adjacent elements, and swaps them if necessary. The next iteration depends on the results of the previous one.
- **Linear Search**: Involves checking each element in a list sequentially until a match is found.

Key Characteristics:

- Execution is done step-by-step.
- Each step is dependent on the previous one.
- No parallel execution.

Parallel Algorithm

A **parallel algorithm** is one where multiple tasks are executed simultaneously across multiple processors or cores. This type of algorithm takes advantage of concurrency to speed up computation, especially for large problems or data sets.

Examples:

- **Matrix Multiplication**: In a parallel implementation, the multiplication of different matrix cells can be performed simultaneously.
- **Parallel Merge Sort**: In parallel Merge Sort, the merge operation can be done concurrently for different subarrays.

Key Characteristics:

- Tasks are divided into sub-tasks that can be executed simultaneously.
- Suitable for problems that can be broken down into independent tasks.
- Requires parallel hardware (e.g., multi-core processors).

Advantages:

- Reduces computation time for large datasets.
- Utilizes multiple processors efficiently.
- Can solve large-scale problems that are too slow for sequential algorithms.

---

4. Heuristic Algorithms

A **heuristic algorithm** is an approach to problem-solving that uses a practical method or shortcut to produce solutions that are good enough for a given problem, especially when finding an optimal solution is too time-consuming or computationally expensive. Heuristic algorithms provide approximate solutions rather than exact ones, which makes them particularly useful for complex or NP-hard problems.

Examples:

- *A Search Algorithm**: Used in pathfinding, where the algorithm uses heuristics to estimate the cost to reach the goal and chooses the best path accordingly.
- **Traveling Salesman Problem (TSP)**: Heuristic methods like the nearest neighbor algorithm can be used to find an approximate solution to TSP, even though the exact solution may be computationally expensive.

- **Simulated Annealing**: A probabilistic technique for approximating the global optimum of a given function.

Key Characteristics:

- Approximate solutions
- Efficient in finding good solutions quickly
- Not guaranteed to find the optimal solution

---

5. Approximate Algorithms

Approximate algorithms are algorithms designed to find near-optimal solutions to complex problems where exact solutions are computationally expensive or infeasible. These algorithms typically provide a solution that is close to the optimal solution, often within a specified error bound.

Examples:

- **Greedy Algorithms**: These algorithms make the best choice at each step with the hope of finding the optimal solution. However, they may not always produce the optimal solution, but a solution that is good enough.
    - **Example**: The Greedy algorithm for the **Knapsack Problem** does not always yield the optimal solution, but it provides a fast and reasonable approximation.
- **Linear Programming Approximation**: Approximation techniques in linear programming can find solutions to complex problems like resource allocation without solving the exact problem.

Key Characteristics:

- Close-to-optimal solutions
- Suitable for complex problems where exact solutions are computationally expensive
- Often used in optimization problems

Conclusion

Algorithms form the backbone of problem-solving in computer science, and their classification into deterministic vs. non-deterministic, divide and conquer, series and parallel, heuristic, and approximate types helps in choosing the right approach for different types of problems. Understanding the properties and applications of these algorithms allows you to design and implement efficient solutions for a wide range of problems in computing, from sorting and searching to complex optimization and approximation tasks.

Lab Works

Here's a set of Java code implementations for the specified lab exercises:

1. Implementations of Different Operations Related to Stack

```
class Stack {
    private int maxSize;
    private int top;
    private int[] stack;

    public Stack(int size) {
        maxSize = size;
        stack = new int[maxSize];
        top = -1;
    }

    // PUSH operation
    public void push(int value) {
        if (top < maxSize - 1) {
            stack[++top] = value;
            System.out.println(value + " pushed to stack");
        } else {
            System.out.println("Stack Overflow");
        }
    }

    // POP operation
    public void pop() {
        if (top >= 0) {
            System.out.println(stack[top--] + " popped from stack");
        } else {
            System.out.println("Stack Underflow");
```

```
        }
    }

    // PEEK operation
    public void peek() {
        if (top >= 0) {
            System.out.println("Top element: " + stack[top]);
        } else {
            System.out.println("Stack is empty");
        }
    }

    public static void main(String[] args) {
        Stack stack = new Stack(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.peek();
        stack.pop();
        stack.peek();
    }
}
```

2. Implementations of Different Operations Related to Linear and Circular Queues

```
class Queue {
    private int maxSize;
    private int front, rear;
    private int[] queue;

    public Queue(int size) {
```

```
    maxSize = size;
    queue = new int[maxSize];
    front = -1;
    rear = -1;
}

// Enqueue operation for Linear Queue
public void enqueue(int value) {
    if (rear < maxSize - 1) {
        if (front == -1) front = 0;
        queue[++rear] = value;
        System.out.println(value + " added to queue");
    } else {
        System.out.println("Queue Overflow");
    }
}

// Dequeue operation for Linear Queue
public void dequeue() {
    if (front == -1) {
        System.out.println("Queue Underflow");
    } else {
        System.out.println(queue[front++] + " removed from queue");
        if (front > rear) {
            front = rear = -1;
        }
    }
}
```

```
// Circular Queue
public void circularEnqueue(int value) {
    if ((rear + 1) % maxSize == front) {
        System.out.println("Queue Overflow");
    } else {
        if (front == -1) front = 0;
        rear = (rear + 1) % maxSize;
        queue[rear] = value;
        System.out.println(value + " added to circular queue");
    }
}

public void circularDequeue() {
    if (front == -1) {
        System.out.println("Queue Underflow");
    } else {
        System.out.println(queue[front] + " removed from circular queue");
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % maxSize;
        }
    }
}

public static void main(String[] args) {
    Queue q = new Queue(5);
    q.enqueue(10);
    q.enqueue(20);
```

```
        q.dequeue();
        q.circularEnqueue(30);
        q.circularDequeue();
    }
}
```

3. Solutions of TOH and Fibonacci Series Using Recursion

```
// Tower of Hanoi
class TowerOfHanoi {
    public void solveTOH(int n, char source, char destination, char auxiliary) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
            return;
        }
        solveTOH(n - 1, source, auxiliary, destination);
        System.out.println("Move disk " + n + " from " + source + " to " + destination);
        solveTOH(n - 1, auxiliary, destination, source);
    }

    public static void main(String[] args) {
        TowerOfHanoi toh = new TowerOfHanoi();
        toh.solveTOH(3, 'A', 'C', 'B');
    }
}

// Fibonacci Series using Recursion
class Fibonacci {
    public int fib(int n) {
        if (n <= 1) {
            return n;
```

```
        }
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] args) {
        Fibonacci fibonacci = new Fibonacci();
        int n = 6;
        System.out.println("Fibonacci of " + n + " is: " + fibonacci.fib(n));
    }
}
```

4. Implementations of Different Operations Related to Linked List (Singly and Doubly Linked List)

```
// Singly Linked List
class SinglyLinkedList {
    Node head;

    static class Node {
        int data;
        Node next;

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

    // Insert at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
```

```
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    // Print the list
    public void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();
    }
}
```

```
// Doubly Linked List
class DoublyLinkedList {
    Node head;

    static class Node {
        int data;
        Node next;
        Node prev;

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

    // Insert at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
            newNode.prev = current;
        }
```

```
}

// Print the list in forward direction
public void printList() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}

// Print the list in reverse direction
public void printReverse() {
    Node current = head;
    while (current != null && current.next != null) {
        current = current.next;
    }
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.prev;
    }
    System.out.println();
}

public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();
    list.insert(10);
    list.insert(20);
```

```
        list.insert(30);
        list.printList();
        list.printReverse();
    }
}
```

5. Implementation of Trees: AVL Trees, Balancing of AVL

// AVL Tree implementation is more complex. Below is a simplified skeleton:

```
class AVLTree {
    class Node {
        int data, height;
        Node left, right;

        public Node(int data) {
            this.data = data;
            height = 1;
        }
    }

    private Node root;

    // Get height of the node
    private int height(Node N) {
        if (N == null) {
            return 0;
        }
        return N.height;
    }
```

```
// Get balance factor
private int getBalance(Node N) {
    if (N == null) {
        return 0;
    }
    return height(N.left) - height(N.right);
}

// Right rotate
private Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    x.right = y;
    y.left = T2;

    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;

    return x;
}

// Left rotate
private Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    y.left = x;
    x.right = T2;
```

```
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    return y;
}

// Insert node and balance AVL
public Node insert(Node node, int data) {
    if (node == null) {
        return new Node(data);
    }

    if (data < node.data) {
        node.left = insert(node.left, data);
    } else if (data > node.data) {
        node.right = insert(node.right, data);
    } else {
        return node;
    }

    node.height = 1 + Math.max(height(node.left), height(node.right));

    int balance = getBalance(node);

    if (balance > 1 && data < node.left.data) {
        return rightRotate(node);
    }
```

```
        if (balance < -1 && data > node.right.data) {
            return leftRotate(node);
        }

        if (balance > 1 && data > node.left.data) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        if (balance < -1 && data < node.right.data) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }

    public static void main(String[] args) {
        AVLTree tree = new AVLTree();
        tree.root = tree.insert(tree.root, 10);
        tree.root = tree.insert(tree.root, 20);
        tree.root = tree.insert(tree.root, 30);
    }
}
```

Note: Implementations for the other exercises (Merge Sort, Searching Techniques, Graph Traversals, Hashing, Heap, etc.) require similar levels of detail. Let me know if you want implementations for each or specific ones.

Syllabus

Course Description

Course Description

This course includes fundamental concept of data structures such as stack, queue, list.

linked list, trees and graph: application of these data structures along with several algorithms.

Course Objectives

The general objective of this course is to provide fundamental concepts of data structures, different algorithms and their implementation.

Unit Contents

1. Introduction to Data Structure : 2 hrs

Definition, Abstract Data Type, Importance of Data structure

2. The Stack : 3 hrs

Introduction, Stack as an ADT, POP and PUSH Operation, Stack Application: Evaluation of Infix, Postfix, and Prefix Expressions, Conversion of Expression.

3. Queue : 3 hrs

Introduction, Queue as an ADT , Primitive Operations in Queue, Linear and Circular Queue and Their Application, Enqueue and Dequeue, Priority Queue

4. List : 2 hrs

Introduction, Static and Dynamic List Structure, Array Implementation of Lists, Queue as a list

5. Linked Lists : 5 hrs

Introduction, Linked List as an ADT, Dynamic Implementation, Insertion & Deletion of Nodes, Linked Stacks and Queues, Doubly Linked Lists and Its Advantages

6. Recursion : 4 hrs

Introduction, Principle of Recursion, Recursion vs. Iteration, Recursion Example: TOH and Fibonacci Series, Applications of Recursion, Search Tree

7. Trees : 5 hrs

Introduction, Basic Operation in Binary tree, Tree Search and Insertion/Deletion, Binary Tree Transversals(pre-order, post-order and in-order), Tree Height, Level and Depth, Balanced Trees: AVL Balanced Trees, Balancing Algorithm, The Huffman Algorithm, Game tree, B-Tree

8. Sorting : 5 hrs

Introduction, Internal and External Sort, Insertion and Selection Sort, Exchange Sort, Bubble and Quick Sort, Merge and Radix Sort, Shell Sort, Binary Sort, Heap Sort as Priority Queue, Efficiency of Sorting, Big'O'Notation.

9. Searching : 5 hrs

Introduction to Search Technique; essential of search, Sequential search, Binary search, Tree search, General search tree, Hashing: Hash function and hash tables, Collision resolution technique, Efficiency comparisons of different search technique.

10. Graphs : 5 hrs

Introduction, Graphs as an ADT, Transitive Closure, Warshall's Algorithm, Types of Graph, Graph Traversal and Spanning Forests, Kruskal's and Round-Robin Algorithms, Shortest- path Algorithm, Greedy Algorithm, DijKstra's Algorithm

11. Algorithms : 5 hrs

Deterministic and Non-deterministic Algorithm, Divide and Conquer Algorithm, Series and Parallel Algorithm, Heuristic and Approximate Algorithms

Laboratory Works

There shall be 10 lab exercises based on C or Java

1. Implementations of different operations related to Stack
2. Implementations of different operations related to linear and circular queues
3. Solutions of TOH and Fibonacci Series using Recursion
4. Implementations of different operations related to linked list: singly and doubly linked
5. Implementation of trees: AVL trees, Balancing of AVL
6. Implementation of Merge sort
7. Implementation of different searching technique: sequential, Tree and Binary
8. Implementation of Graphs: Graph traversals
9. Implementation of Hashing
10. Implementations of Heap