Data Structures and Algorithms: Complete Solutions

Module I: Introduction and Arrays

Question 1: Define data structures. Classify various types with examples.

Solution: Data structures are systematic ways of organizing and storing data in a computer so that it can be accessed and modified efficiently. They dictate how data is stored, related, and manipulated.

Data structures are broadly classified into two main categories:

1. **Primitive Data Structures:** These are basic data types that are directly operated upon by machine-level instructions. They hold single values.
    - **Examples:** int (integer), float (floating-point number), char (character), boolean (true/false).
2. **Non-Primitive Data Structures:** These are more complex structures derived from primitive data structures. They are used to store collections of data. They can be further categorized:
    - **Linear Data Structures:** Elements are arranged sequentially, and each element has a predecessor and successor (except the first and last).
        - **Examples:**
            - **Arrays:** A collection of fixed-size, homogeneous elements stored in contiguous memory locations (e.g., [10, 20, 30, 40]).
            - **Linked Lists:** A collection of nodes where each node contains data and a pointer to the next node (e.g., Node1 -> Node2 -> Node3).
            - **Stacks:** A LIFO (Last In, First Out) structure where elements are added and removed from the same end (e.g., (top) 3 -> 2 -> 1 (bottom)).
            - **Queues:** A FIFO (First In, First Out) structure where elements are added at one end and removed from the other (e.g., (front) 1 -> 2 -> 3 (rear)).
    - **Non-Linear Data Structures:** Elements are not arranged sequentially. They represent hierarchical or networked relationships.
        - **Examples:**
            - **Trees:** Hierarchical structures where each node has zero or more child nodes, with a single root node (e.g., a file system directory structure).
            - **Graphs:** A collection of nodes (vertices) and edges that connect pairs of nodes, representing relationships (e.g., a social network graph).

Question 2: What are the advantages and disadvantages of arrays?

Solution: Advantages of Arrays:

1. **Fast Random Access:** Elements can be accessed directly using their index in constant time, irrespective of the array size. This is due to their contiguous memory allocation.
2. **Memory Locality:** Elements are stored in contiguous memory locations, which leads to better cache performance and faster processing for sequential access patterns.
3. **Simplicity:** Arrays are conceptually simple and easy to understand and implement.
4. **Efficient for Fixed-Size Data:** They are very efficient when the number of elements is known in advance and doesn't change frequently.

Disadvantages of Arrays:

1. **Fixed Size:** Once an array is declared with a specific size, its size cannot be changed during runtime. If more elements are needed, a new, larger array must be created, and all elements copied, which is inefficient.
2. **Inefficient Insertions and Deletions:** Adding an element in the middle or deleting an element requires shifting all subsequent elements to maintain contiguity, leading to time complexity (where is the number of elements).
3. **Memory Waste:** If an array is declared with a large size but not all slots are used, memory is wasted. Conversely, if it's too small, resizing is costly.
4. **Homogeneous Data:** Arrays typically store elements of the same data type. While some languages support arrays of objects, the basic concept is for uniform data types.

Question 3: Describe the structure of a one-dimensional array. How are elements accessed and stored?

Solution: Structure of a One-Dimensional Array: A one-dimensional array is a linear collection of elements of the same data type (e.g., all integers, all characters, etc.). These elements are stored in contiguous (adjacent) memory locations. This contiguous storage is a fundamental characteristic that allows for efficient direct access. Each element in the array is uniquely identified by an integer index, typically starting from 0 for the first element.

How Elements are Stored: When an array is declared, a block of memory is reserved that is large enough to hold all its elements sequentially. For example, if an integer array of size 5 is declared, and each integer takes 4 bytes, then 20 bytes of continuous memory will be allocated. The elements are laid out one after another in this block.

- Array[0] will be at the starting memory address.
- Array[1] will be at starting address + size_of_element.
- Array[i] will be at starting address + i size_of_element.

How Elements are Accessed: Elements in a one-dimensional array are accessed using their **index** (or subscript). The index specifies the position of the element within the array.

- If an array arr has n elements, its indices typically range from 0 to n-1.
- To access the first element, you use arr[0].
- To access the fifth element, you use arr[4].
- To access the element at index i, you use arr[i].

This direct access is highly efficient (constant time,) because the memory address of any element can be calculated directly using its base address (the address of the first element), its index, and the size of each element.

Question 4: How is an array initialized and accessed in memory? Explain **the concept of index calculation.**

Solution: Array Initialization: Arrays can be initialized in various ways depending on the programming language.

1. **Declaration with Size and Default Values:** When an array is declared with a specific size, memory is allocated, and elements are often initialized to default values (e.g., 0 for numeric types, null for objects, garbage values in C/C++ if not explicitly initialized).
   - Example (Java): int[] numbers = new int[5]; // Initializes all 5 elements to 0
2. **Declaration with Initializer List:** Values can be provided directly at the time of declaration, and the compiler determines the size.
   - Example (C++/Java): int numbers[] = {10, 20, 30, 40, 50}; // Size is implicitly 5

Array Access in Memory and Index Calculation: Arrays are stored in a contiguous block of memory. This contiguity is key to their efficient access.

- **Base Address:** Every array has a base address, which is the memory address of its first element (at index 0). Let's denote this as .

- **Element Size:** Each element in the array has a fixed size in bytes (e.g., an integer might be 4 bytes, a character 1 byte). Let this be .
- **Index:** To access any element at a specific position, we use its index, say . Indices typically start from 0.

The memory address of an element at a given index is calculated using the following formula:

Concept of Index Calculation: This formula illustrates the concept of index calculation. When you request Array[i], the system doesn't search for the element; instead, it directly calculates its memory location.

For example, consider an integer array myArray where:

- BaseAddress = 1000 (starting memory address)
- SizeOfElement = 4 bytes (for an integer)

To access myArray[0]:

To access myArray[3]:

This direct computation allows for (constant time) access, making arrays very fast for retrieving elements once their index is known.

Question 5: Write a program to perform matrix addition using 2D arrays.

Solution:

```
# Module I: Introduction and Arrays
# Question 5: Write a program to perform matrix addition using 2D arrays.

def matrix_addition(matrix1, matrix2):
    """
    Performs addition of two matrices.

    Args:
        matrix1 (list of list of int/float): The first matrix.
        matrix2 (list of list of int/float): The second matrix.

    Returns:
        list of list of int/float: The resultant matrix after addition,
                                   or None if matrices have incompatible dimensions.
    """
    rows1 = len(matrix1)
    cols1 = len(matrix1[0]) if rows1 > 0 else 0

    rows2 = len(matrix2)
    cols2 = len(matrix2[0]) if rows2 > 0 else 0

    # Check if matrices have compatible dimensions for addition
    if rows1 != rows2 or cols1 != cols2:
        print("Error: Matrices must have the same dimensions for addition.")
        return None

    # Initialize a result matrix with zeros
    result_matrix = [[0 for _ in range(cols1)] for _ in range(rows1)]

    # Iterate through rows
    for i in range(rows1):
        # Iterate through columns
        for j in range(cols1):
            result_matrix[i][j] = matrix1[i][j] + matrix2[i][j]

    return result_matrix

def print_matrix(matrix):
    """
    Prints a matrix in a readable format.
    """
    if matrix is None:
        return
    for row in matrix:
        print(" ".join(map(str, row)))

# Example Usage:
print("--- Matrix Addition Example ---")

# Define two matrices
matrix_a = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix_b = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]

print("Matrix A:")
print_matrix(matrix_a)

print("\nMatrix B:")
print_matrix(matrix_b)

# Perform matrix addition
sum_matrix = matrix_addition(matrix_a, matrix_b)

if sum_matrix:
    print("\nResultant Matrix (A + B):")
    print_matrix(sum_matrix)

# Example with incompatible dimensions
matrix_c = [[1, 2], [3, 4]]
matrix_d = [[5, 6, 7], [8, 9, 0]]

print("\n--- Example with Incompatible Dimensions ---")
print("Matrix C:")
print_matrix(matrix_c)
print("\nMatrix D:")
print_matrix(matrix_d)
matrix_addition(matrix_c, matrix_d) # This will print an error message
```

Module II: Stacks and Queues

Question 1: Define a stack. Explain push and pop operations with an example.

Solution: A **stack** is a linear data structure that follows a particular order in which operations are performed. This order is known as **LIFO (Last In, First Out)**. This means the last element added to the stack is the first one to be removed. Think of a stack of plates: you can only add a new plate on top, and you can only remove the top-most plate.

The two primary operations associated with a stack are:

1. **Push Operation:**
    - **Definition:** The push operation is used to add an element to the top of the stack.
    - **Behavior:** When an element is pushed, it becomes the new top element. If the stack is full (in implementations with a fixed size), an "overflow" condition may occur.
    - **Time Complexity:** (constant time).
2. **Pop Operation:**
    - **Definition:** The pop operation is used to remove the topmost element from the stack.
    - **Behavior:** When an element is popped, it is removed, and the element that was previously below it becomes the new top. If the stack is empty, an "underflow" condition may occur.
    - **Time Complexity:** (constant time).

Example: Let's consider a stack initially empty.

- **Initial Stack:** []

1. **Push 10:**
    - Operation: push(10)
    - Result: [10] (10 is now at the top)
2. **Push 20:**
    - Operation: push(20)
    - Result: [10, 20] (20 is now at the top)
3. **Push 30:**
    - Operation: push(30)
    - Result: [10, 20, 30] (30 is now at the top)
4. **Pop:**
    - Operation: pop()
    - Result: [10, 20] (30 is removed, 20 is now at the top). The popped element is 30.
5. **Push 40:**
    - Operation: push(40)
    - Result: [10, 20, 40] (40 is now at the top)
6. **Pop:**
    - Operation: pop()
    - Result: [10, 20] (40 is removed, 20 is now at the top). The popped element is 40.

This example clearly demonstrates the LIFO principle: 30 was pushed after 20, and 30 was popped before 20 (and 10).

Question 2: Differentiate between stack and queue in terms of structure and usage.

Solution:

Feature	Stack	Queue
Principle	LIFO (Last In, First Out)	FIFO (First In, First Out)
Structure	Elements are added and removed from the same end, called the **top**.	Elements are added at one end (rear/tail) and removed from the other end (front/head).
Operations	**Push:** Adds an element to the top. **Pop:** Removes an element from the top. **Peek/Top:** Accesses the top element.	**Enqueue:** Adds an element to the rear. **Dequeue:** Removes an element from the front. **Peek/Front:** Accesses the front element.
Analogy	A stack of plates, a pile of books.	A waiting line, a ticket counter queue.
Usage/Typical **Applications**	Function call management (call stack). Undo/Redo functionality in applications. Expression evaluation (infix to postfix). Backtracking algorithms. Browser history (back button).	Task scheduling (e.g., print jobs). Breadth-First Search (BFS) in graphs. CPU scheduling. Data buffering/streaming. Handling asynchronous data.

Question 3: What are the different types of queues? Explain with diagrams.

Solution: Queues are linear data structures following the FIFO (First In, First Out) principle. Here are the different types:

1. **Simple/Linear Queue:**
   - **Description:** This is the most basic form where elements are added at the rear and removed from the front. Once an element is dequeued, the space it occupied cannot be reused unless the entire queue is shifted. This can lead to "queue full" even if there are empty slots at the beginning (front).
   - **Diagram:**

```
FRONT -> [E1] [E2] [E3] [E4] <- REAR
         (Added first)      (Added last)
```

   - **Limitation:** Inefficient use of memory if frequent enqueues and dequeues occur, as the front pointer keeps advancing.
2. **Circular Queue:**
   - **Description:** Overcomes the space limitation of a linear queue by treating the array as a circular structure. When the rear reaches the end of the array, it wraps around to the beginning if there are empty slots. This allows for more efficient use of the allocated memory.

- **Diagram:**

```
   [ ] [ ] [E4]
   ^        |
   |        V
FRONT <---- [E1] [E2] [E3]
(REAR just wrapped around to add E4)

(Imagine a circular arrangement where the end connects to the beginning)
```

- **Advantage:** Efficient use of memory and avoids the "queue full" problem of linear queues unless truly full.

3. **Priority Queue:**

- **Description:** Unlike simple queues, elements in a priority queue are not served strictly based on their arrival time. Instead, each element has a priority associated with it, and elements with higher priority are served before elements with lower priority. If priorities are the same, they are usually served based on FIFO.
- **Diagram:**

```
(Highest Priority First)
[Prio 1, E1] <- [Prio 2, E2] <- [Prio 2, E3] <- [Prio 3, E4]
            (E1 dequeued before E2/E3/E4)
```

- **Usage:** Operating system process scheduling, network packet processing.

4. **Deque (Double-Ended Queue):**

- **Description:** A deque is a more generalized queue where elements can be added or removed from *both* the front and the rear ends. It offers more flexibility than a standard queue or stack.
- **Diagram:**

```
FRONT <-> [E1] <-> [E2] <-> [E3] <-> REAR
(Can add/remove from either end)
```

- **Usage:** Can be used to implement both stacks and queues, and algorithms like the A-star search.

Question 4: Explain the role of stacks **in expression evaluation and parsing.**

Solution: Stacks play a crucial role in both expression evaluation and parsing due to their LIFO (Last In, First Out) nature, which naturally handles operator precedence and operand ordering.

1. Role in Expression Evaluation: Stacks are primarily used for:

- **Infix** to Postfix/Prefix **Conversion:** Infix expressions (e.g., A + B C) are difficult for computers to evaluate directly due to varying operator precedence and associativity. Stacks are used to convert these into postfix (e.g., A B C +) or prefix (e.g., + A B C) forms, which are easier to evaluate.
    - **Mechanism:** When converting, operands are directly outputted. Operators are pushed onto an operator stack based on their precedence. Higher precedence operators are popped and outputted before lower precedence ones when encountered. Parentheses are used to control the pushing and popping of operators.
- **Postfix Expression Evaluation:** Once an expression is in postfix form, it can be evaluated efficiently using a single operand stack.
    - **Mechanism:**
        1. Scan the postfix expression from left to right.
        2. If an operand is encountered, push it onto the operand stack.

3. If an operator is encountered, pop the required number of operands (usually two for binary operators) from the stack, perform the operation, and push the result back onto the stack.
4. After scanning the entire expression, the final result will be the only element left on the stack.

Example (Postfix Evaluation of 2 3 5 +**):**

1. Read 2: Push 2. Stack: [2]
2. Read 3: Push 3. Stack: [2, 3]
3. Read : Pop 3, Pop 2. Compute 2 3 = 6. Push 6. Stack: [6]
4. Read 5: Push 5. Stack: [6, 5]
5. Read +: Pop 5, Pop 6. Compute 6 + 5 = 11. Push 11. Stack: [11] Result: 11.

2. Role in Parsing (Syntax Analysis): In compilers and interpreters, parsing involves checking if a sequence of tokens (like keywords, identifiers, operators) conforms to the grammatical rules of a language. Stacks are fundamental in **syntax analysis** (specifically, bottom-up parsing techniques like LR parsing).

- **Mechanism:** Parsers often use a stack to store grammar symbols (terminals and non-terminals) and states. As input tokens are read, they are shifted onto the stack. When the top of the stack matches a right-hand side of a grammar production rule, a "reduce" operation is performed, replacing the matched symbols with the left-hand side non-terminal. This process continues until the entire input is reduced to the start symbol of the grammar.
- **Handling Nested Structures:** Stacks are ideal for handling nested constructs like parentheses (), curly braces {}, or square brackets []. When an opening bracket is encountered, it's pushed onto the stack. When a closing bracket is found, the corresponding opening bracket is popped. If they don't match, or if a closing bracket is found with an empty stack, a syntax error is flagged.
- **Recursion Simulation:** Many parsing algorithms are inherently recursive. Stacks implicitly manage the call stack for recursive function calls, helping to keep track of the current state of parsing and where to return after processing a sub-structure.

In summary, stacks provide the necessary LIFO behavior to manage the order of operations and hierarchical relationships essential for correctly evaluating expressions and verifying the syntax of code.

Question 5: Write a program to implement a circular queue using arrays.

Solution:

```
# Module II: Stacks and Queues
# Question 5: Write a program to implement a circular queue using arrays.

class CircularQueue:
    """
    Implements a circular queue using a Python list (array).
    """

    def __init__(self, capacity):
        """
        Initializes the circular queue with a given capacity.
        The queue will store up to 'capacity' elements.
        """
        self.capacity = capacity
        # The underlying array/list to store queue elements.
        # We add 1 to capacity because one slot is kept empty to distinguish
        # between full and empty states.
        self.queue = [None]  (capacity + 1)
        # Front points to the first element (or the slot BEFORE the first element)
        self.front = 0
        # Rear points to the last element
        self.rear = 0
        self.size = 0 # Current number of elements in the queue

    def is_empty(self):
        """
        Checks if the queue is empty.
        In a circular queue, front == rear indicates an empty queue.
        """
        return self.front == self.rear

    def is_full(self):
        """
        Checks if the queue is full.
        In a circular queue, (rear + 1) % (capacity + 1) == front indicates a full queue.
        We reserve one slot to differentiate full from empty.
        """
        return (self.rear + 1) % (self.capacity + 1) == self.front

    def enqueue(self, item):
        """
        Adds an element to the rear of the queue.
        """
        if self.is_full():
            print("Queue is full! Cannot enqueue.")
            return False

        self.rear = (self.rear + 1) % (self.capacity + 1)
        self.queue[self.rear] = item
        self.size += 1
        print(f"Enqueued: {item}")
        return True

    def dequeue(self):
        """
        Removes and returns the element from the front of the queue.
        """
        if self.is_empty():
            print("Queue is empty! Cannot dequeue.")
            return None

        self.front = (self.front + 1) % (self.capacity + 1)
        item = self.queue[self.front]
        self.queue[self.front] = None # Optional: Clear the dequeued slot
        self.size -= 1
        print(f"Dequeued: {item}")
        return item

    def peek(self):
        """
        Returns the front element without removing it.
        """
        if self.is_empty():
            print("Queue is empty! No element to peek.")
            return None

        # The actual front element is at (self.front + 1) in our implementation
        return self.queue[(self.front + 1) % (self.capacity + 1)]

    def get_size(self):
        """
        Returns the current number of elements in the queue.
        """
        return self.size
    def display(self):
        """
        Displays the current elements in the queue.
        """
        if self.is_empty():
            print("Queue is empty.")
            return

        print("Queue elements (front to rear): ", end="")
        current_index = (self.front + 1) % (self.capacity + 1)
        count = 0
        while count < self.size:
            print(self.queue[current_index], end=" ")
            current_index = (current_index + 1) % (self.capacity + 1)
            count += 1
        print()

# Example Usage:
print("--- Circular Queue Example ---")
cq = CircularQueue(capacity=3) # Capacity for 3 elements

cq.display()
print(f"Is Empty: {cq.is_empty()}")
print(f"Is Full: {cq.is_full()}")

cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30) # Queue is now full (3 elements)
cq.display()
print(f"Size: {cq.get_size()}")
print(f"Is Full: {cq.is_full()}")

cq.enqueue(40) # Attempt to enqueue when full

print(f"Front element: {cq.peek()}")

cq.dequeue() # Dequeue 10
cq.display()
print(f"Size: {cq.get_size()}")

cq.enqueue(40) # Now there's space, 40 is added (wraps around)
cq.display()
print(f"Size: {cq.get_size()}")
print(f"Is Full: {cq.is_full()}") # Should be full again

cq.dequeue() # Dequeue 20
cq.dequeue() # Dequeue 30
cq.dequeue() # Dequeue 40
cq.display()
print(f"Size: {cq.get_size()}")
print(f"Is Empty: {cq.is_empty()}")

cq.dequeue() # Attempt to dequeue when empty
```

Module III: Linked Lists

Question 1: Define a singly linked list. Write an algorithm to insert a node at the end.

Solution: Definition of a Singly Linked List: A **singly linked list** is a linear data structure composed of a sequence of **nodes**. Unlike arrays, elements in a linked list are not stored in contiguous memory locations. Each node in a singly linked list consists of two parts:

1. **Data Part:** Stores the actual value or information.
2. **Pointer (or Reference) Part:** Stores the memory address (or reference) of the **next** node in the sequence.

The first node in the list is called the **head**, and it points to the beginning of the list. The last node in the list has its pointer part pointing to NULL (or None in Python), indicating the end of the list.

Algorithm to Insert a Node at the End of a Singly Linked List:

This algorithm assumes the existence of a Node structure (or class) with data and next attributes, and a head pointer for the linked list.

Algorithm: insertAtEnd(head, newData)

1. **Create a New Node:**
    - Create a new node, let's call it newNode.
    - Set newNode.data = newData.
    - Set newNode.next = NULL (since it will be the last node).
2. **Check if List is Empty:**
    - If head is NULL (the list is empty):
        - Set head = newNode.
        - Return head.
3. **Traverse to the Last Node:**
    - If the list is not empty, start from the head.
    - Create a temporary pointer, say current = head.
    - Iterate through the list until current.next is NULL. This means current is currently pointing to the last node.
        - While current.next is not NULL:
            - current = current.next
4. **Link the New Node:**
    - Once the loop terminates, current points to the last node of the original list.
    - Set current.next = newNode.
5. **Return Head (unchanged unless the list was initially empty):**
    - Return head.

Pseudocode:

```
// Define a Node structure
Structure Node:
   data
   next (pointer to Node)
```

```
Function insertAtEnd(head, newData):
  // 1. Create a New Node
  newNode = new Node()
  newNode.data = newData
  newNode.next = NULL

  // 2. Check if List is Empty
  If head is NULL:
    head = newNode
    Return head

  // 3. Traverse to the Last Node
  current = head
  While current.next is not NULL:
    current = current.next

  // 4. Link the New Node
  current.next = newNode

  // 5. Return Head
  Return head
```

Question 2: Explain the differences between singly, doubly, and circular linked lists with diagrams.

Solution: The primary distinction among these linked list types lies in how their nodes are linked, specifically concerning the number and direction of pointers.

1. **Singly Linked List:**

   - **Description:** Each node contains data and a single pointer that points to the *next* node in the sequence. Traversal is only possible in one direction (forward). The last node's pointer is NULL.
   - **Diagram:**

```
HEAD -> [Data|Next] -> [Data|Next] -> [Data|NULL]
         Node 1        Node 2        Node 3 (Tail)
```

   - **Characteristics:** Simple to implement, memory efficient (one pointer per node).

2. **Doubly Linked List:**

   - **Description:** Each node contains data and two pointers: one pointing to the *next* node and another pointing to the *previous* node. This allows for bidirectional traversal (forward and backward). The prev pointer of the first node and the next pointer of the last node are NULL.
   - **Diagram:**

```
HEAD -> [NULL|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|NULL]
         Node 1              Node 2              Node 3 (Tail)
```

   - **Characteristics:** More flexible for insertions/deletions and bidirectional traversal. Requires more memory (two pointers per node).

3. **Circular Linked List:**

   - **Description:** Similar to a singly or doubly linked list, but the last node points back to the first node, forming a closed loop. There is no NULL pointer to mark the end of the list, which can sometimes simplify certain operations (e.g., iterating through all elements starting from any point).
   - **Types:** Can be singly circular or doubly circular.
   - **Diagram (Singly Circular Linked List):**

```
HEAD -> [Data|Next] -> [Data|Next] -> [Data|Next]
         Node 1        Node 2        Node 3
         ^                          |
         |__________________________| (Node 3 points back to Node 1)
```

- **Characteristics:** No NULL values, continuous traversal possible, useful for managing buffers that wrap around or round-robin scheduling.

Question 3: What are the advantages of using a linked list over an array?

Solution: Linked lists offer several significant advantages over arrays, particularly in scenarios where dynamic size and efficient modifications are crucial:

1. **Dynamic Size:**
    - **Linked List:** They can grow or shrink in size dynamically at runtime as needed. There's no need to pre-allocate a fixed amount of memory. Nodes are created and destroyed as elements are added or removed.
    - **Array:** Arrays have a fixed size defined at the time of declaration. If the capacity is exceeded, a new larger array must be created, and all existing elements must be copied to the new array, which is an expensive operation.
2. **Efficient Insertions and Deletions:**
    - **Linked List:** Inserting a new node or deleting an existing node (once the position is found or the preceding node is known) requires only changing a few pointers. This operation takes constant time. For example, to insert in the middle, you change the next pointer of the previous node and the next pointer of the new node.
    - **Array:** Inserting or deleting an element in the middle of an array requires shifting all subsequent elements to make space or close the gap, respectively. This takes linear time, where is the number of elements.
3. **No Memory Waste (Less Fragmentation):**
    - **Linked List:** Memory for nodes is allocated only when needed (dynamically, typically from the heap). This means no memory is wasted for unused pre-allocated space. Nodes can be stored non-contiguously, making efficient use of available memory blocks.
    - **Array:** If an array is declared with a large size but not fully utilized, the unused portion of the allocated memory block is wasted.
4. **Flexible Memory Allocation:**
    - **Linked List:** Nodes can be stored anywhere in memory as long as their addresses are linked correctly. This allows for more flexible memory usage patterns.
    - **Array:** Requires a single, large contiguous block of memory, which might be hard to find if the array is very large, especially in memory-constrained environments.

These advantages make linked lists a preferred choice for implementing dynamic data structures like stacks, queues, and for scenarios where frequent insertions and deletions are expected, such as in operating system task schedulers or dynamic memory allocation.

Question 4: Compare arrays and linked lists in terms of memory usage, performance, and flexibility.

Solution: Let's compare arrays and linked lists across key aspects:

1. **Memory Usage:**
    - **Arrays:**

        - **Pros:** Efficient memory usage per element for storing data itself, as there's no overhead for pointers. Excellent **cache locality** because elements are contiguous, leading to faster access for sequential operations as the CPU can prefetch data.
        - **Cons:** Can lead to **memory fragmentation** if large contiguous blocks are hard to find. Potential **memory waste** if declared size is much larger than actual usage. Fixed size can lead to expensive reallocations and copies if the size needs to increase.
    - **Linked Lists:**
        - **Pros:** Uses memory dynamically (allocates only as needed), leading to less wasted space if the exact number of elements is unknown. No need for contiguous memory blocks, reducing fragmentation issues.
        - **Cons:** Higher **memory overhead** per element due to the storage required for pointers (each node needs memory for data *plus* one or more pointers). Poorer **cache locality** because nodes can be scattered throughout memory, requiring more memory fetches.
2. **Performance:**
    - **Arrays:**
        - **Random Access (by index):** Excellent, constant time. Direct address calculation.
        - **Insertion/Deletion (middle):** Poor, linear time, as elements need to be shifted.
        - **Traversal:** Good, linear time. Fast due to cache locality.
    - **Linked Lists:**
        - **Random Access (by index):** Poor, linear time. Requires sequential traversal from the head to reach the -th element.
        - **Insertion/Deletion (middle, if position known):** Excellent, constant time. Only pointer adjustments are needed. However, finding the position might take .
        - **Traversal:** Good, linear time. Slower than arrays due to lack of cache locality.
3. **Flexibility:**
    - **Arrays:**
        - **Size:** Inflexible; fixed size determined at creation. Resizing is expensive.
        - **Structure:** Simple, contiguous block. Good for direct mapping to mathematical vectors/matrices.
        - **Data Types:** Typically homogeneous (all elements of the same type).
    - **Linked Lists:**
        - **Size:** Highly flexible; dynamically resizable. Can grow or shrink as elements are added or removed without explicit reallocation.
        - **Structure:** More complex with individual nodes and pointers.
        - **Data Types:** Can be homogeneous or heterogeneous (if storing pointers to generic objects), depending on language implementation.

Summary Table:

Feature	**Arrays**	**Linked Lists**
Size	Fixed	Dynamic

Memory Usage	Data-dense, good cache, potential waste	Pointer overhead, poor cache, no waste
Random Access	(Excellent)	(Poor)
Insert/Delete	(Poor, requires shifting)	(Excellent, if node/prev known)
Traversal	(Good, due to cache)	(Good, but slower due to scattered nodes)
Memory Req.	Contiguous block	Dispersed nodes

Question 5: Discuss the traversal and search operations in a singly linked list.

Solution: In a singly linked list, elements are not stored contiguously, and each node only points to the *next* node. This structure dictates how traversal and search operations are performed.

1. Traversal Operation: Definition: Traversal refers to the process of visiting each node in the linked list exactly once, typically starting from the head node and moving sequentially to the end.

Algorithm:

1. **Start from the Head:** Initialize a temporary pointer (e.g., current) to the head of the linked list.
2. **Iterate until Null:** While current is not NULL (meaning you haven't reached the end of the list):
    - **Process the Current Node:** Perform the desired operation on the data of the current node (e.g., print its value, perform a calculation).
    - **Move to the Next Node:** Update current = current.next. This moves the pointer to the subsequent node in the list.
3. **End of Traversal:** The loop terminates when current becomes NULL, indicating that all nodes have been visited.

Pseudocode:

```
Function traverseList(head):
    current = head
    While current is not NULL:
        Print current.data  // Or perform any other operation
        current = current.next
```

Time Complexity: The time complexity for traversing a singly linked list is , where is the number of nodes. This is because, in the worst case, every node must be visited once.

2. Search Operation: Definition: The search operation in a singly linked list involves finding a node that contains a specific data value (the "key").

Algorithm:

1. **Start from the Head:** Initialize a temporary pointer (e.g., current) to the head of the linked list.
2. **Iterate and Compare:** While current is not NULL:
    - **Check Data:** Compare the data in the current node with the key you are searching for.
    - **Match Found:** If current.data equals key:
        - The element is found. Return true (or the current node, or its index).

- **Move to Next Node:** If no match, update current = current.next.

3. **End of Search:** If the loop finishes and current becomes NULL (meaning the end of the list is reached) without finding a match:
    - The element is not found in the list. Return false.

Pseudocode:

```
Function searchList(head, key):
    current = head
    While current is not NULL:
        If current.data equals key:
            Return true // Element found
        current = current.next
    Return false // Element not found
```

Time Complexity: The time complexity for searching in a singly linked list is . In the worst case (element is at the end or not present), you might have to visit every node. In the best case (element is at the head), it's . On average, it's . This linear time complexity is a consequence of the sequential access nature of linked lists; random access is not possible.

Module IV: Trees

Question 1: Define a binary tree. Explain its types and applications.

Solution: Definition of a Binary Tree: A **binary tree** is a hierarchical data structure in which each node has at most two children, typically referred to as the **left child** and the **right child**. It's a non-linear data structure where elements are organized in a parent-child relationship. The topmost node is called the **root** node, and nodes with no children are called **leaf nodes**.

Types of Binary Trees:

1. **Full Binary Tree (Strictly Binary Tree):**
    - Every node has either 0 or 2 children. No node has only one child.
    - Example: A node has two children, or it's a leaf.
2. **Complete Binary Tree:**
    - All levels are completely filled, except possibly the last level.
    - If the last level is not full, its nodes are filled from left to right.
    - Example: A tree where all nodes are as far left as possible.
3. **Perfect Binary Tree:**
    - A binary tree in which all internal nodes have two children, and all leaf nodes are at the same level (same depth).
    - This is a special case of both a full and a complete binary tree.
    - Example: A fully balanced tree where every level is entirely filled.
4. **Degenerate/Skewed Binary Tree:**
    - Every parent node has only one child. It essentially behaves like a linked list.
    - Example: A tree where all nodes form a single chain, either to the left or to the right.

Applications of Binary Trees:

1. **Expression Trees:** Used to represent arithmetic or logical expressions. Internal nodes are operators, and leaf nodes are operands. This allows for easy evaluation of expressions.
2. **Binary Search Trees (BSTs):** A special type of binary tree that maintains a sorted order (left child < parent < right child). This enables efficient searching, insertion, and deletion of elements (on average). Used in database indexing and dictionary implementations.
3. **File Systems:** Although often represented as general trees, hierarchical file systems (directories, subdirectories, files) can be conceptualized using tree structures.
4. **Compilers:** Syntax trees (parse trees) are binary trees that represent the syntactic structure of source code, which is then used for semantic analysis and code generation.
5. **Heaps:** A specific type of complete binary tree that satisfies the heap property (parent is always greater than/less than its children). Used in priority queues and heap sort algorithms.
6. **Decision Trees:** In machine learning, decision trees are used for classification and regression tasks, where each internal node represents a test on an attribute, and each leaf node represents a class label or a decision.
7. **Data Compression:** Huffman coding trees are binary trees used for data compression, where more frequent characters have shorter codes.

Question 2: Compare binary tree and binary search tree.

Solution: Both binary trees and binary search trees (BSTs) are hierarchical data structures. A BST is a special kind of binary tree, distinguished by a specific ordering property.

Feature	Binary Tree	Binary Search Tree (BST)
Definition	Each node has at most two children (left and right). There are no ordering constraints on data.	A binary tree with a special ordering property: for any given node, all values in its **left subtree** are *less than* the node's value, and all values in its **right subtree** are *greater than* the node's value.
Ordering Property	No inherent ordering of data.	Strict ordering of data relative to the parent node. Duplicates are generally not allowed or handled specifically.
Structure	Any structure is valid as long as it adheres to the "at most two children" rule. Can be unbalanced, skewed, etc.	Structure is constrained by the ordering property. Can still be unbalanced if data is inserted in sorted/reverse-sorted order.
Insertion	New nodes can be inserted almost anywhere (e.g., as the next available child or based on specific application logic).	Insertion requires traversing the tree to find the correct position while maintaining the BST property (smaller to left, larger to right).
Search	Requires traversing the entire tree (e.g., BFS or DFS) in the worst case to find an element, time complexity.	Highly efficient search. Traverses a single path from the root down. Average time complexity is for balanced trees, for skewed trees.
Use Case	Representing hierarchical data (e.g., expression trees, general family trees), syntax trees in compilers.	Efficient storage and retrieval of sorted data (e.g., dictionary, set implementations, database indexing, symbol tables).
Example Values	Parent: 10 Left Child: 5 Right Child: 20 *(This structure is valid, but no ordering required)*	Parent: 10 Left Child: 5 (must be < 10) Right Child: 20 (must be > 10) *(This structure obeys the BST property)*
Balance	Concept of balance is relevant for performance, but not strictly required for its definition.	Balance is crucial for performance. Unbalanced BSTs degenerate to linked lists, losing their efficiency for . Self-balancing BSTs (AVL, Red-Black trees) maintain balance automatically.

Question 3: What are tree traversal techniques? Explain inorder, preorder, and postorder traversal with an example.

Solution: Tree traversal techniques are methods for visiting (processing) each node in a tree data structure exactly once. Since trees are non-linear, there are different standard ways to navigate them. For binary trees, the three primary depth-first traversal methods are Inorder, Preorder, and Postorder.

Let's use the following **example binary tree**:

```
      10
     /  \
    5    15
   / \
  3   7
```

1. Inorder Traversal (Left -> Root -> Right):

- **Definition:** In this traversal, we first visit the left subtree, then the root node, and finally the right subtree. This process is applied recursively to each subtree.

- **Behavior:** For a Binary Search Tree (BST), an inorder traversal yields the elements in **sorted order**.
- **Process:**
    - Recursively traverse the left subtree.
    - Visit (process) the current node (root).
    - Recursively traverse the right subtree.
- **Example Walkthrough:**
    - Go left from 10 to 5.
    - Go left from 5 to 3.
    - 3 has no left/right children. Visit 3. Output: 3
    - Return to 5. Visit 5. Output: 3, 5
    - Go right from 5 to 7.
    - 7 has no left/right children. Visit 7. Output: 3, 5, 7
    - Return to 10. Visit 10. Output: 3, 5, 7, 10
    - Go right from 10 to 15.
    - 15 has no left/right children. Visit 15. Output: 3, 5, 7, 10, 15
- **Result:** 3, 5, 7, 10, 15

2. Preorder Traversal (Root -> Left -> Right):

- **Definition:** In this traversal, we first visit the root node, then recursively traverse the left subtree, and finally recursively traverse the right subtree.
- **Behavior:** Useful for creating a copy of the tree or for prefix expressions in expression trees.
- **Process:**
    - Visit (process) the current node (root).
    - Recursively traverse the left subtree.
    - Recursively traverse the right subtree.
- **Example Walkthrough:**
    - Visit 10. Output: 10
    - Go left from 10 to 5.
    - Visit 5. Output: 10, 5
    - Go left from 5 to 3.
    - Visit 3. Output: 10, 5, 3
    - Return to 5. 3 has no children. Go right from 5 to 7.
    - Visit 7. Output: 10, 5, 3, 7
    - Return to 10. 7 has no children. Go right from 10 to 15.
    - Visit 15. Output: 10, 5, 3, 7, 15
- **Result:** 10, 5, 3, 7, 15

3. Postorder Traversal (Left -> Right -> Root):

- **Definition:** In this traversal, we first recursively traverse the left subtree, then recursively traverse the right subtree, and finally visit the root node.
- **Behavior:** Useful for deleting a tree (deleting children before the parent) or for postfix expressions in expression trees.
- **Process:**
    - Recursively traverse the left subtree.
    - Recursively traverse the right subtree.
    - Visit (process) the current node (root).
- **Example Walkthrough:**
    - Go left from 10 to 5.
    - Go left from 5 to 3.
    - 3 has no left/right children. Return to 5.
    - Go right from 5 to 7.
    - 7 has no left/right children. Return to 5.
    - Visit 5 (after left/right subtrees of 5 are processed). Output: 3, 7, 5
    - Return to 10. Go right from 10 to 15.
    - 15 has no left/right children. Return to 10.
    - Visit 10 (after left/right subtrees of 10 are processed). Output: 3, 7, 5, 15, 10
- **Result:** 3, 7, 5, 15, 10

Question 4: Write an algorithm for inorder traversal using recursion.

Solution: The Inorder traversal of a binary tree visits the nodes in the order: Left Subtree -> Root -> Right Subtree. Recursion is a natural fit for tree traversals because the definition of a tree (a root, a left subtree, and a right subtree, where subtrees are also trees) is inherently recursive.

Algorithm: inorderTraversal(node)

1. **Base Case:**
    - If node is NULL (or None in Python), return. This means we've reached an empty subtree or gone past a leaf node.
2. **Recursive Step - Left Subtree:**
    - Call inorderTraversal(node.left). This will recursively visit all nodes in the left subtree.
3. **Process Root:**
    - Visit (process) the node itself. This typically means printing its data, but could be any operation.
4. **Recursive Step - Right Subtree:**
    - Call inorderTraversal(node.right). This will recursively visit all nodes in the right subtree.

Pseudocode:

```
// Assume a Node structure exists:
// Structure Node:
//     data
//     left (pointer to Node)
//     right (pointer to Node)
```

```
Function inorderTraversal(node):
    // 1. Base Case: If the current node is null, we've gone too far or hit an empty tree.
    If node is NULL:
        Return

    // 2. Recursively traverse the left subtree
    inorderTraversal(node.left)

    // 3. Visit the current node (Root)
    Print node.data  // Or perform other operations on node.data

    // 4. Recursively traverse the right subtree
    inorderTraversal(node.right)
```

Example Trace (using the example tree from Q3):

```
        10
       /  \
      5    15
     / \
    3   7
```

~~~
```inorderTraversal(10)`
    `inorderTraversal(5)`
        `inorderTraversal(3)`
            `inorderTraversal(NULL)` -> Return
            Print `3`
            `inorderTraversal(NULL)` -> Return
        Print `5`
        `inorderTraversal(7)`
            `inorderTraversal(NULL)` -> Return
            Print `7`
            `inorderTraversal(NULL)` -> Return
    Print `10`
    `inorderTraversal(15)`
        `inorderTraversal(NULL)` -> Return
        Print `15`
        `inorderTraversal(NULL)` -> Return
~~~

Output: 3 5 7 10 15

Question 5: Describe the use of trees in hierarchical data representation.

Solution:
Trees are exceptionally well-suited for representing hierarchical data because their structure naturally mirrors parent-child relationships, sub-categories, and nested levels. In a tree, the root node represents the highest level in the hierarchy, and its children represent the next level of sub-categories or components. Each child can, in turn, become a parent to its own children, forming deeper levels of the hierarchy.

Here's how trees are used in hierarchical data representation:

1. File Systems: This is one of the most common and intuitive examples.
   The root directory (e.g., `/` in Unix-like systems, `C:\` in Windows) is the root node of the tree.
   Directories (folders) are internal nodes that can have children (sub-directories or files).
   Files are typically leaf nodes (they don't have children).
   The path from the root to any file or directory traces its position within the hierarchy (e.g., `/home/user/documents/report.pdf`).

2. Organizational Charts:
   The CEO or head of the organization is the root node.
   Department heads are children of the CEO.

Team leads are children of department heads, and so on, down to individual employees (leaf nodes).
This clearly shows reporting structures and departmental divisions.

3. XML/HTML/JSON Document Structures (Parse Trees/DOM Trees):
   When an XML, HTML, or JSON document is parsed, it's often represented internally as a tree.
   Each tag or object becomes a node.
   Nested tags/objects/arrays become children of their parent tag/object.
   The Document Object Model (DOM) in web browsers is a tree-based representation of an HTML or XML document, allowing programmatic access and manipulation of its structure and content.

4. Biological Taxonomies:
   The classification of living organisms (Kingdom, Phylum, Class, Order, Family, Genus, Species) is inherently hierarchical and can be represented as a tree, showing evolutionary relationships.

5. Family Trees/Genealogies:
   Individuals are nodes, and parent-child relationships are represented by edges, forming a tree (though often with more complex structures like directed acyclic graphs to handle marriages and multiple parents).

6. Table of Contents / Outlines:
   The main title is the root.
   Chapters are children of the main title.
   Sections are children of chapters, and subsections are children of sections.

In all these scenarios, the tree structure efficiently models the "contains," "belongs to," or "sub-category of" relationships, making it easy to navigate, search, and manage data based on its position within the hierarchy. Operations like finding all descendants of a node (e.g., all files in a directory) or finding the parent of a node (e.g., the containing folder) are natural fits for tree algorithms.

Module V: Graphs and Sorting

Question 1: Explain the adjacency matrix and adjacency list representations of graphs.

Solution:
Graphs are fundamental data structures representing relationships between entities. There are two common ways to represent a graph's connections: the adjacency matrix and the adjacency list.

1. Adjacency Matrix:
Description: An adjacency matrix is a square 2D array (matrix) where the rows and columns represent the vertices (nodes) of the graph.
Representation:
If there are `V` vertices, the matrix will be `V x V`.
For an unweighted graph, `Matrix[i][j] = 1` if there is an edge from vertex `i` to vertex `j`, and `0` otherwise.
For a weighted graph, `Matrix[i][j]` could store the weight of the edge from `i` to `j`, or `infinity`/`0` if no edge exists.
For an undirected graph, the matrix is symmetric (i.e., `Matrix[i][j] = Matrix[j][i]`).
Example:
Consider a graph with 3 vertices (0, 1, 2) and edges (0,1), (1,2), (2,0).
```
  0 1 2
0[0 1 1]
1[1 0 1]
2[1 1 0]
```
(Note: (0,2) is added for symmetry for undirected graph, so 0 is connected to 1 and 2, etc.)

Advantages:
Easy to check for edge existence: Checking if an edge exists between two vertices `i` and `j` is $O(1)$ by simply looking up `Matrix[i][j]`.
Easy to add/remove edges: $O(1)$ operation by changing a single matrix entry.
Convenient for dense graphs (graphs with many edges).
Disadvantages:
Space Inefficient: Requires $O(V^2)$ space, even for sparse graphs (graphs with few edges), where most entries would be 0.
Inefficient for finding neighbors: To find all neighbors of a vertex, you need to iterate through an entire row or column ($O(V)$ time).

2. Adjacency List:
Description: An adjacency list represents a graph as an array (or list) of linked lists (or dynamic arrays). The index of the array corresponds to a vertex, and each linked list at that index contains the vertices adjacent to it.
Representation:
An array of size `V`.
`AdjList[i]` contains a list of all vertices `j` such that there is an edge from `i` to `j`.
For weighted graphs, each item in the list could be a pair: `(neighbor_vertex, weight)`.
Example:
Consider the same graph with 3 vertices (0, 1, 2) and edges (0,1), (1,2), (2,0).
```
0: -> 1 -> 2
1: -> 0 -> 2
2: -> 0 -> 1
```
(For an undirected graph, if (0,1) exists, then 1 must also be in 0's list and 0 in 1's list)

Advantages:
Space Efficient: Requires $O(V + E)$ space, where `E` is the number of edges. This is much more efficient for sparse graphs.
Efficient for finding neighbors: Finding all neighbors of a vertex `i` is efficient, as you just traverse the linked list at `AdjList[i]`, which takes $O(\text{degree}(i))$ time.
Disadvantages:
Less efficient for checking edge existence: Checking if an edge exists between `i` and `j` requires traversing `AdjList[i]`, which can take up to $O(V)$ time in the worst case.

Less efficient to add/remove edges in some cases: Can be $O(\text{degree}(i))$ to find and remove an edge if not using a hash set for the list.

Choice of Representation:
Adjacency Matrix is preferred for dense graphs (E is close to V^2) and when frequent checks for edge existence are required.
Adjacency List is preferred for sparse graphs (E is much less than V^2) and when frequent traversal of neighbors is required (e.g., in BFS/DFS algorithms).

Question 2: Compare BFS and DFS with respect to data structure usage and traversal order.

Solution:
Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental algorithms for traversing or searching graph (and tree) data structures. They differ significantly in how they explore nodes and the auxiliary data structures they employ.

| Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental graph traversal techniques that differ significantly in their approach and use cases. BFS explores a graph level by level, meaning it visits all neighboring nodes at the current depth before moving on to the next level. This makes BFS behave like ripples spreading outward from a starting point. It uses a queue (FIFO) to store and process nodes, ensuring that nodes are visited in the order they are discovered. BFS is widely used for finding the shortest path in unweighted graphs, identifying connected components, powering web crawlers, and performing social network analysis such as finding friends of friends. Its time complexity is O(V + E), and it requires O(V) space in the worst case because of the queue and the storage of visited nodes. Importantly, BFS guarantees the shortest path in terms of number of edges.
In contrast, DFS explores a graph by moving as deep as possible along one branch before backtracking, similar to navigating a maze by following one path to its end and then trying others. DFS uses either a stack (LIFO) or recursion, which internally uses the call stack. DFS is useful for tasks such as detecting cycles, performing topological sorting, solving puzzles like mazes and Sudoku, and identifying strongly connected components. Like BFS, its time complexity is O(V + E), but space usage depends on the recursion or stack depth, with a worst case of O(V). Unlike BFS, DFS does not guarantee finding the shortest path. BFS is usually implemented iteratively with a queue, while DFS can be implemented either iteratively with a stack or more commonly through recursion.

Summary:
BFS is like a wave spreading out, ensuring all nodes at a shallower "depth" are visited before moving deeper. It uses a queue to maintain this order.
DFS is like a drill going deep down one path as far as it can, then retracting and trying another path. It uses a stack (or recursion's implicit stack) to keep track of the current path and backtracking points.---

Question 3: What is hashing? Explain different collision resolution techniques.

Solution:

Hashing is a technique or a process of mapping arbitrary size data (key) to a fixed-size value (hash value or hash code) using a mathematical function called a hash function. The hash value is then typically used as an index to store the original data in a data structure known as a hash table (or hash map).

Purpose: The primary goal of hashing is to enable very fast (ideally $O(1)$ average time) insertion, deletion, and retrieval of data. It converts a key (which could be a name, ID, etc.) into an address where the data is stored.
Hash Function: A good hash function distributes keys uniformly across the hash table, minimizes collisions, and is computationally efficient.

Collision Resolution Techniques:
A collision occurs when two different keys hash to the same index (hash value) in the hash table. Since each slot can hold only one item (initially), a strategy is needed to handle such situations. These strategies are called collision resolution techniques. The two main categories are:

1. Open Addressing (Closed Hashing):
In open addressing, when a collision occurs, the system probes (searches) for the next available empty slot within the same hash table. Different probing methods determine the sequence of slots to check.

a) Linear Probing:
Mechanism: If the initial hash index `h` is occupied, it checks `h+1`, then `h+2`, and so on (modulo table size) until an empty slot is found.
Problem: Suffers from primary clustering, where long runs of occupied slots form. This causes future insertions and searches to take longer.
Example: If `key1` hashes to index 5, and `key2` also hashes to 5 (collision), `key2` will be placed at 6 (if 5 is occupied), then `key3` might hash to 5 and go to 7, extending the cluster.

b) Quadratic Probing:
Mechanism: If the initial hash index `h` is occupied, it checks `h + 1^2`, then `h + 2^2`, `h + 3^2`, and so on (modulo table size).
Improvement: Reduces primary clustering by spreading out probes.
Problem: Suffers from secondary clustering, where keys that hash to the same initial position follow the same probe sequence.

c) Double Hashing:
Mechanism: Uses two hash functions. If the first hash function `h1(key)` results in a collision, a second hash function `h2(key)` is used to determine the step size for probing. The probe sequence is `(h1(key) + i  h2(key)) % table_size`.
Improvement: Provides the best collision resolution among open addressing techniques, as it minimizes both primary and secondary clustering by creating a different probe sequence for each key.

2. Separate Chaining (Open Hashing):
In separate chaining, instead of probing for another slot in the main table, each slot in the hash table is designed to hold a pointer to a linked list (or another data structure like a dynamic array or balanced BST). When a collision occurs, the new key-value pair is simply added to the linked list at the calculated hash index.

Mechanism:
Each bucket (index) in the hash table is a "chain" (typically a linked list).
When `key` hashes to index `i`, the `(key, value)` pair is added to the linked list at `table[i]`.
To search for a key, hash it to find the bucket, then traverse the linked list at that bucket.
Example:
If both `keyA` and

4. Compare and contrast bubble sort and selection sort in terms of time complexity and usage.

Bubble Sort

- Working:
  Repeatedly compares adjacent elements and swaps them if they are in the wrong order. After each pass, the largest element “bubbles up” to its correct position.
- Time Complexity:
  - Best case: O(n) (when the array is already sorted, using an optimized version)
  - Average case: O(n^2)
  - Worst case: O(n^2)
- Space Complexity:
  - O(1) (in-place)
- Usage:
  - Useful for small datasets.
  - Good for educational purposes to understand basic sorting ideas.
  - Not preferred for large inputs due to slow performance.

Selection Sort

- Working:
  In each pass, finds the minimum element from the unsorted part and places it in its correct position by swapping.
- Time Complexity:
  - Best case: $O(n^2)$
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$
- Space Complexity:
  - O(1) (in-place)
- Usage:
  - Works well when memory writes should be minimized (performs fewer swaps than bubble sort).
  - Good for small datasets.
  - Not suitable for large datasets.

Comparison Table

Feature	Bubble Sort	Selection Sort
Basic Operation	Compare adjacent pairs and swap	Find minimum element and swap
Best Time	O(n) (optimized)	$O(n^2)$
Average/Worst Time	$O(n^2)$	$O(n^2)$
Number of Swaps	High (can be many swaps)	Low ($\leq$ n swaps)
Usage	Learning, small datasets	Small datasets, when swap cost is high

5. Write an algorithm for merge sort and explain its divide-and-conquer approach.

Merge Sort Algorithm

Algorithm: MERGE-SORT(A, left, right)

```
1. If left < right
2.     Find the middle:
   mid = (left + right) / 2
3.     Call MERGE-SORT(A, left, mid)
4.     Call MERGE-SORT(A, mid + 1, right)
5.     Call MERGE(A, left, mid, right)
6. End if
```

Algorithm: MERGE(A, left, mid, right)

```
1. Create two temporary arrays:
       • L = A[left … mid]
       • R = A[mid+1 … right]
2. Set index pointers i = 0, j = 0, and k = left
3. While both arrays have elements:
```

    - If L[i] ≤ R[j], set A[k] = L[i], increment i
    - Else set A[k] = R[j], increment j
    - Increment k
4. Copy any remaining elements from L
5. Copy any remaining elements from R

Divide-and-Conquer Approach in Merge Sort

1. Divide:
   Split the unsorted array into two halves until each subarray has only one element.
2. Conquer:
   Recursively sort the two halves.
3. Combine:
   Merge the two sorted halves to produce one fully sorted array.

Complexity

- Time Complexity:
    - Best: O(n log n)
    - Average: O(n log n)
    - Worst: O(n log n)
- Space Complexity:
    - O(n) (needs extra memory for merging)
- Stable Sorting: Yes
- Usage:
    - Suitable for large datasets.
    - Works well for linked lists.
    - Used in external sorting (like sorting data from disk).