---

1. Introduction to Data Structures and Algorithms

Basic Terminologies

- **Data Structure**: A way to organize and store data for efficient access and modification.
    - Example: Arrays, Linked Lists, Stacks, Queues, Trees, Graphs.
- **Elementary Data Organizations**: Data can be organized linearly (arrays, linked lists) or non-linearly (trees, graphs).
- **Data Structure Operations**:
    - **Insertion**: Adding a new element.
    - **Deletion**: Removing an element.
    - **Traversal**: Accessing each element exactly once.
    - **Searching**: Finding an element.
    - **Sorting**: Arranging elements in a specific order.

Analysis of Algorithms

- **Time Complexity**: Amount of time taken by an algorithm to run as a function of input size.
- **Space Complexity**: Amount of memory used by an algorithm.
- **Asymptotic Notations**:
    - **Big-O (O)**: Upper bound (worst-case complexity).
    - **Big-Ω (Ω)**: Lower bound (best-case complexity).
    - **Big-Θ (Θ)**: Tight bound (average-case complexity).
- **Time-Space Tradeoff**: Reducing time complexity may increase space complexity and vice versa.

---

2. Searching Techniques

Linear Search

Algorithm:
python
Copy

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index if found

```

- `    return -1  # Return -1 if not found`
- **Time Complexity**: O(n) (worst case).

Binary Search

Algorithm:
python
Copy
```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
```

- `        return -1`
- **Time Complexity**: O(log n) (worst case).
- **Prerequisite**: The array must be sorted.

---

3. Stacks and Queues

Stack (ADT)

- **Operations**:
    - **Push**: Add an element to the top.
    - **Pop**: Remove the top element.
    - **Peek**: View the top element.
- **Applications**:
    - Expression conversion (infix to postfix).
    - Evaluation of postfix expressions.
    - Backtracking algorithms.

Algorithm for Infix to Postfix Conversion:
python
Copy
```
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack = []
    output = []
    for char in expression:
        if char.isalnum():
```

```
            output.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()
        else:
            while stack and precedence.get(stack[-1], 0) >= precedence.get(char, 0):
                output.append(stack.pop())
            stack.append(char)
    while stack:
        output.append(stack.pop())
```

- ```
  return ''.join(output)
  ```

Queue (ADT)

- **Types**:
  - **Simple Queue**: FIFO (First In First Out).
  - **Circular Queue**: Rear connects to the front.
  - **Priority Queue**: Elements are dequeued based on priority.
- **Operations**:
  - **Enqueue**: Add an element to the rear.
  - **Dequeue**: Remove an element from the front.

Algorithm for Circular Queue:

python
Copy

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = self.rear = -1

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            print("Queue is full")
        elif self.front == -1:
            self.front = self.rear = 0
            self.queue[self.rear] = item
        else:
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = item
```
```

```
    def dequeue(self):
        if self.front == -1:
            print("Queue is empty")
        elif self.front == self.rear:
            temp = self.queue[self.front]
            self.front = self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
```

- return temp

---

4. Linked Lists

Singly Linked List

- **Operations**:

Insertion at the beginning:
python
Copy

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

def insert_at_beginning(head, data):
    new_node = Node(data)
    new_node.next = head
    head = new_node
```

    - return head

Deletion:
python
Copy

```
def delete_node(head, key):
    temp = head
    if temp and temp.data == key:
        head = temp.next
        temp = None
```

```
    return head
  while temp:
    if temp.data == key:
      break
    prev = temp
    temp = temp.next
  if not temp:
    return head
  prev.next = temp.next
  temp = None
```

- return head

Doubly Linked List

- **Operations**:

Insertion at the beginning:
python
Copy

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

def insert_at_beginning(head, data):
  new_node = Node(data)
  new_node.next = head
  if head:
    head.prev = new_node
  head = new_node
```

- return head

---

5. Practice Questions

Group-A (Very Short Answer Type)

1. **Hash Function**: A function that maps data to a fixed-size value (e.g., for indexing in hash tables).
2. **ADT**: Abstract Data Type is a theoretical model for data structures (e.g., Stack, Queue).
3. **Overflow Condition**: FRONT = REAR + 1 for a **Circular Queue**.

4. **Node**: A basic unit in a data structure (e.g., in Linked Lists, Trees).
5. **Sibling Node**: Nodes at the same level in a tree.
6. **Directed Graph**: A graph with edges having a direction.
7. **Quick Sort Worst Case**: $O(n^2)$.
8. **Hash Collision**: When multiple keys map to the same bucket.
9. **Time Complexity**: Measure of time taken by an algorithm.
10. **Queue Configuration**: 3 deletions and 3 additions.

Group-B (Short Answer Type)

Bubble Sort Algorithm:

```
python
Copy
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:

1.                  arr[j], arr[j+1] = arr[j+1], arr[j]
```

2. **Min-Priority Queue**:
    - **Insert**: Add an element while maintaining the heap property.
    - **Decrease Key**: Reduce the value of a key and adjust the heap.

Group-C (Long Answer Type)

Binary Search Algorithm:

```
python
Copy
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

1.      return -1
```

2. **Infix to Postfix Conversion**:
    - Use a stack to convert (7+3)*5 to 7 3 + 5 *.
    - Evaluate postfix using a stack.

---

6. Sorting Algorithms

Objective of Sorting

- Rearrange elements in a list or array in a specific order (ascending or descending).
- Improve the efficiency of other algorithms (e.g., searching, merging).

Properties of Sorting Algorithms

1. **Time Complexity**: How fast the algorithm runs.
2. **Space Complexity**: How much extra memory the algorithm uses.
3. **Stability**: Whether the algorithm maintains the relative order of equal elements.
4. **In-Place**: Whether the algorithm requires extra memory or not.

---

1. Selection Sort

- **Algorithm**:
    - Repeatedly find the minimum element from the unsorted part and swap it with the first unsorted element.

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

```

- ```
          arr[i], arr[min_idx] = arr[min_idx], arr[i]
  ```
- **Time Complexity**: $O(n^2)$ (worst, average, and best case).
- **Space Complexity**: O(1) (in-place).
- **Stability**: Not stable.

  ```

2. Bubble Sort

- **Algorithm**:
    - Repeatedly swap adjacent elements if they are in the wrong order.

Python Code:
python
Copy
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:

-                   arr[j], arr[j+1] = arr[j+1], arr[j]
```

- **Time Complexity**: O(n²) (worst and average case), O(n) (best case if already sorted).
- **Space Complexity**: O(1) (in-place).
- **Stability**: Stable.

---

3. Insertion Sort

- **Algorithm**:
    - Build the sorted array one element at a time by inserting each new element into its correct position.

Python Code:
python
Copy
```
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
```

- **Time Complexity**: O(n²) (worst and average case), O(n) (best case if already sorted).
- **Space Complexity**: O(1) (in-place).
- **Stability**: Stable.

---

4. Quick Sort

- **Algorithm**:
    - Choose a pivot element and partition the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right. Recursively sort the sub-arrays.

Python Code:
python
Copy
```
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)`
- **Time Complexity**: O(n log n) (average case), O(n²) (worst case if the pivot is poorly chosen).
- **Space Complexity**: O(log n) (due to recursion stack).
- **Stability**: Not stable.

---

5. Merge Sort

- **Algorithm**:
    - Divide the array into two halves, recursively sort each half, and then merge the two sorted halves.

Python Code:
python
Copy
```
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
- **Time Complexity**: O(n log n) (worst, average, and best case).
- **Space Complexity**: O(n) (extra space for merging).
- **Stability**: Stable.

---

6. Heap Sort

- **Algorithm**:
    - Build a max-heap from the array, repeatedly extract the maximum element, and place it at the end of the array.

Python Code:
python
Copy

```
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)
- **Time Complexity**: O(n log n) (worst, average, and best case).
- **Space Complexity**: O(1) (in-place).
- **Stability**: Not stable.

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No

Performance Comparison of Sorting Algorithms

Practice Questions on Sorting

Group-A (Very Short Answer Type)

1. What is the best-case time complexity of Bubble Sort?
   Answer: O(n) (when the array is already sorted).
2. What is the worst-case time complexity of Quick Sort?
   Answer: $O(n^2)$ (when the pivot is poorly chosen).
3. Which sorting algorithm is stable among Quick Sort, Merge Sort, and Heap Sort?
   Answer: Merge Sort.
4. What is the space complexity of Merge Sort?
   Answer: O(n).

5. What is the time complexity of Heap Sort?
   Answer: O(n log n).

Group-B (Short Answer Type)

1. Write the algorithm for Insertion Sort.
   Answer: See the Python code above.
2. Compare the performance of Bubble Sort and Quick Sort.
   Answer: Bubble Sort has $O(n^2)$ time complexity in the worst and average cases, while Quick Sort has O(n log n) in the average case but $O(n^2)$ in the worst case. Quick Sort is generally faster.

Group-C (Long Answer Type)

1. Explain the Merge Sort algorithm with an example.
   Answer: Merge Sort divides the array into two halves, recursively sorts each half, and then merges the two sorted halves. Example:
   Input: [38, 27, 43, 3, 9, 82, 10]
   Output: [3, 9, 10, 27, 38, 43, 82]
2. Write a Python program to implement Heap Sort.
   Answer: See the Python code above.

Tips for Exam Preparation

1. Practice writing algorithms and code snippets.
2. Understand the time and space complexity of each operation.
3. Solve previous year's question papers.
4. Focus on key topics: Stacks, Queues, Linked Lists, Searching, and Sorting.

Good luck with your exam! 🚀