black N White

NAME	
ROLL NUMBER	
SEMESTER	**2nd**
COURSE CODE	**DCA1207**
COURSE NAME	**BCA**
Subject Name	**DATA STRUCTURES**

SET - I

Q.1) What do you understand by Algorithm Complexity? Discuss Time and Space Complexity in detail by taking suitable examples

Answer .:-

Algorithm Complexity

Algorithm complexity refers to the measure of the amount of **resources** an algorithm uses to solve a given computational problem. The two primary resources considered are **time** and **space**. Complexity helps in analyzing how efficient an algorithm is in terms of its execution and memory usage, especially when the input size increases.

Understanding algorithm complexity allows developers to compare different algorithms and select the most optimal one for their needs.

1. Time Complexity

Time complexity represents the total time taken by an algorithm to complete its execution, depending on the size of the input. It is generally expressed using **Big O notation (O)**, which describes the upper bound of the time an algorithm can take in the worst-case scenario.

Common Time Complexities:

- **O(1)** – Constant time (e.g., accessing an array element)
- **O(n)** – Linear time (e.g., traversing a list)
- **O(n²)** – Quadratic time (e.g., bubble sort)
- **O(log n)** – Logarithmic time (e.g., binary search)

Example:

```
for(int i = 0; i < n; i++) {
   cout << i;
}
```

In this example, the loop runs n times. Hence, the time complexity is **O(n)**.

Binary Search Example:

```
int binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;
   while(low <= high) {
      int mid = (low + high) / 2;
      if(arr[mid] == key)
         return mid;
      else if(arr[mid] < key)
         low = mid + 1;
      else
         high = mid - 1;
   }
   return -1;
}
```

This binary search has a time complexity of **O(log n)** because the search space is divided by 2 after each iteration.

2. Space Complexity

Space complexity refers to the amount of memory an algorithm uses relative to the input size. It includes memory for variables, function calls, and data structures.

Common Space Complexities:

- **O(1)** – Constant space
- **O(n)** – Linear space

Example:

```
void printArray(int arr[], int n) {
   for(int i = 0; i < n; i++) {
      cout << arr[i];
   }
}
```

Here, no extra space is used apart from the input array and loop variables. Thus, space complexity is **O(1)**.

Recursive Example:

```
int factorial(int n) {
   if(n == 0) return 1;
   return n * factorial(n - 1);
}
```

This recursive function uses **O(n)** space due to the call stack created during each recursive call.

Algorithm complexity plays a vital role in determining how efficient a program is. While **time complexity** focuses on the execution time, **space complexity** deals with memory usage. Understanding both helps in writing optimized, scalable, and effective programs for real-world applications.

Q.2) Write an algorithm to find a particular number in an array and replace it with some other value.

Answer .:-

Problem Statement:
Given an array of n elements, find a particular number (target) and replace all its occurrences with another number (new value).

Steps:

1. **Start**
2. Input the size of the array n
3. Declare an array arr[n]

4. Input n elements into the array
5. Input the number to be **found** (let's say target)
6. Input the number to **replace with** (let's say newValue)
7. For i from 0 to n-1, repeat steps 8–9
8. If arr[i] == target then
9. Set arr[i] = newValue
10. End loop
11. Print the updated array
12. **Stop**

Example:

- Input:
  Array: [3, 5, 7, 5, 9]
  Target: 5
  New Value: 0

- Output:
  Updated Array: [3, 0, 7, 0, 9]

Q.3) Explain the working of a Queue data structure. What are its applications in real-world scenarios?

Answer .:-

A **Queue** is a linear data structure that follows the **FIFO** principle – **First In, First Out**. This means that the element inserted first will be removed first, just like people standing in a line: the one who comes first is served first.

Working of a Queue

A queue works through two basic operations:

1. **Enqueue**: This operation adds an element to the **rear** (or end) of the queue.
2. **Dequeue**: This operation removes an element from the **front** of the queue.

Internally, a queue is often implemented using arrays or linked lists. Two pointers are typically used:

- **Front**: Points to the first element in the queue.

- **Rear**: Points to the last element in the queue.

  Example:

  Let's say we insert elements 10, 20, and 30:

- Queue after Enqueue operations: Front → 10 → 20 → 30 ← Rear

  Now, if we perform one Dequeue:

- Queue becomes: Front → 20 → 30 ← Rear (10 is removed)

  Types of Queue:

1. **Simple Queue** – Basic FIFO structure.
2. **Circular Queue** – Connects the rear end back to the front to efficiently use space.
3. **Priority Queue** – Elements are served based on priority, not just the order of insertion.
4. **Deque (Double-Ended Queue)** – Insertion and deletion can occur at both ends.

Queue Operations in Pseudocode:

- **Enqueue(item)**:
  - If the queue is full → show overflow
  - Else:
    - Add item at the rear
    - Increase rear pointer
- **Dequeue()**:
  - If the queue is empty → show underflow
  - Else:
    - Remove item from front
    - Increase front pointer

Real-World Applications of Queue

1. **Printer Queue**:
   When multiple documents are sent to a printer, they are printed in the order they were received.
2. **Customer Service Systems**:
   Call centers use queues to manage customer support, where calls are answered in the order they arrive.
3. **Operating Systems**:
   CPU scheduling uses queues to manage processes in multitasking environments.

4. **Traffic Management**:
   Vehicles waiting at a traffic signal follow the queue structure; the first one to arrive leaves first.
5. **Data Streaming**:
   In multimedia streaming, data packets are queued and processed in order to maintain flow.
6. **Breadth-First Search (BFS)** in Graphs:
   BFS uses a queue to explore nodes level-by-level.
7. **Simulation Systems**:
   Queues are used to model real-world systems such as bank lines or hospital patient processing.

Conclusion

Queues are essential in both theoretical and practical computer science. Their FIFO nature makes them ideal for situations where order matters and elements must be processed in the same sequence they arrive. Understanding queue operations and their use cases helps in solving many real-world and programming problems effectively.

SET - II

Q.4) What is a linked list and its types? Discuss the benefits of using them over array in detail.

Answer .:-

A **linked list** is a **linear data structure** used to store a sequence of elements, where each element (called a **node**) contains two parts:

1. **Data** – The actual value stored
2. **Pointer (or link)** – A reference to the next node in the sequence

Unlike arrays, linked lists **do not store elements in contiguous memory locations**. Instead, each node points to the next, forming a chain-like structure.

Types of Linked Lists

1. **Singly Linked List**
   Each node has a pointer to the **next** node only. Traversal is possible in one direction, from the head (start) to the end.

Structure:

[Data|Next] → [Data|Next] → [Data|NULL]

2. **Doubly Linked List**
   Each node contains **two pointers** – one pointing to the **next** node and one to the **previous**. This allows traversal in **both directions**.

Structure:

NULL ← [Prev|Data|Next] ↔ [Prev|Data|Next] → NULL

3. **Circular Linked List**
   In this type, the last node points back to the **first node**. It can be singly or doubly circular.

Structure:

[Data|Next] → [Data|Next] → ... → [Data|First]

Benefits of Linked List over Arrays

1. **Dynamic Memory Allocation**
   - Arrays have a fixed size and need memory to be allocated beforehand.
   - Linked lists are **dynamic**. Memory is allocated only when needed, reducing wastage.
2. **Efficient Insertion and Deletion**
   - In arrays, inserting or deleting an element (especially in the middle) requires **shifting** other elements, which is costly (O(n)).
   - In linked lists, insertion/deletion is **faster (O(1))** if the position is known, as it involves just changing the links.
3. **No Wastage of Memory**
   - Arrays may have unused slots (if the actual data is less than declared size).
   - Linked lists use **only the memory** required for actual data.
4. **Implementation of Advanced Data Structures**
   - Many complex structures like **stacks**, **queues**, **graphs**, and **hash tables** use linked lists in their implementation.
5. **Ease of Growing Lists**
   - Expanding an array can be costly as it may require creating a new array and copying old data.

- Linked lists can be **easily expanded** by adding new nodes anywhere.

Limitations of Linked List (Compared to Array)

- No direct access: You can't access elements by index as in arrays (arr[i]).
- Slightly higher memory usage: Each node needs to store a pointer along with data.
- Slower traversal: To access a node, you may need to traverse from the beginning.

Conclusion

Linked lists are an essential data structure offering **flexibility**, **efficient memory usage**, and **fast insertions/deletions**. While arrays provide faster random access, linked lists are superior for applications where data size changes frequently or memory management is critical.

Q.5) What is a doubly circular queue? Write an algorithm to display the contents of the circular queue.

Answer .:-

A **Doubly Circular Queue** is a **hybrid data structure** that combines features of a **doubly linked list** and a **circular queue**.

- It is **circular** because the **last node is connected to the first**, forming a loop.
- It is **doubly linked**, meaning **each node has two pointers**: one to the **next node** and one to the **previous node**.

This allows:

- Efficient insertion and deletion from **both ends** (front and rear).
- **Traversal** in both directions (forward and backward).

Structure of a Node

Each node has:

[Prev | Data | Next]

And it connects like:

← [Prev|Data|Next] ⇄ [Prev|Data|Next] ⇄ [Prev|Data|Next] →

↑ ↓

Tail ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←

Algorithm to Display Contents of a Doubly Circular Queue

Assumptions:

- front points to the first node of the queue
- Each node has data, next, and prev

Algorithm:

Step 1: Start

Step 2: If front == NULL

Display "Queue is empty"

Exit

Step 3: Set temp = front

Step 4: Repeat

→ Display temp.data

→ Set temp = temp.next

Until temp == front

Step 5: Stop

Explanation:

- The algorithm starts at the front of the queue.
- It uses a loop to move through each node using the next pointer.
- It continues displaying elements until it returns to the front node, ensuring it prints all nodes **exactly once** in a circular fashion.

Example Output:

If the queue contains 10 → 20 → 30 → 40, the output will be:

Queue contents: 10 20 30 40

Use Cases:

- CPU scheduling (multi-directional rotation)
- Undo/redo operations
- Task management in operating systems

Q.6) Write an algorithm for Merge Sort and explain its divide-and-conquer approach.

Answer .:-

Merge Sort is a popular **sorting algorithm** that uses the **divide-and-conquer** technique. It breaks the problem into smaller sub-problems, solves them independently, and then combines their results.

Divide and Conquer Strategy in Merge Sort

1. **Divide**:
   The array is divided into two halves until each sub-array has only one element.
2. **Conquer**:
   Each small sub-array is sorted individually (since a single element is always sorted).
3. **Combine (Merge)**:
   The sorted sub-arrays are then merged back together in a sorted manner.

This recursive process continues until the whole array is sorted.

Algorithm for Merge Sort

Let A be the input array, and p and r be the starting and ending indices of the array segment.

MergeSort(A, p, r)

1. If p < r:
2. q = (p + r) / 2 // Middle index
3. MergeSort(A, p, q) // Sort left half
4. MergeSort(A, q+1, r) // Sort right half
5. Merge(A, p, q, r) // Merge both halves

Merge(A, p, q, r)

1. Let n1 = q - p + 1
2. Let n2 = r - q
3. Create two temporary arrays L[1...n1] and R[1...n2]
4. Copy data from A[p...q] into L[1...n1]
5. Copy data from A[q+1...r] into R[1...n2]
6. Set i = 0, j = 0, k = p
7. While i < n1 and j < n2:

    If L[i] <= R[j]:

        A[k] = L[i]

        i = i + 1

    Else:

        A[k] = R[j]

        j = j + 1

    k = k + 1
8. Copy any remaining elements of L[]
9. Copy any remaining elements of R[]

Example:

Given array: [38, 27, 43, 3, 9, 82, 10]

- Divide:
  → [38, 27, 43, 3] and [9, 82, 10]
  → Further divide into single elements
- Merge:
  → [27, 38, 43], [3, 9, 10, 82], and finally
  → [3, 9, 10, 27, 38, 43, 82]

Advantages of Merge Sort

- **Stable sort** (does not change the order of equal elements)
- **Guaranteed O(n log n)** time complexity

- Ideal for sorting **linked lists** and **large datasets**

Time Complexity

- **Best, Average, and Worst Case**: O(n log n)
- **Space Complexity**: O(n) (due to temporary arrays)

Merge Sort is a powerful and efficient sorting technique. Its divide-and-conquer nature makes it suitable for large datasets and parallel computing. Though it requires extra memory space, its consistent performance makes it reliable and widely used in real-world applications.