NAME	
ROLL NUMBER	
PROGRAM	**MASTER OF COMPUTER APPLICATIONS (MCA)**
SEMESTER	**2nd**
COURSE CODE	**DCA6204**
COURSE NAME	**ADVANCED DATA STRUCTURES**

SET - I

Q.1.a) Explain different types of data structures in detail

Answer:- Data structures are fundamental building blocks for organizing data in computer programs. Choosing the right one for your needs is crucial for efficient storage and retrieval.

Linear Data Structures:

- **Arrays:** Imagine a fixed-size container with labeled slots. Each slot stores data of the same type, and you access elements using their index (like a slot number). Arrays offer fast random access (retrieving any element by its index) but can be slow for insertions or deletions in the middle, as other elements might need to be shifted.
- **Linked Lists:** Think of train cars linked together. Each car (node) holds data and a pointer to the next car. Linked lists are flexible in size and allow for easier insertions/deletions at any point, but random access is slower as you need to traverse the list car by car to find a specific element.
- **Stacks:** LIFO (Last-In-First-Out) principle, like a stack of plates. You can only add/remove elements from the top. Stacks are great for keeping track of function calls, implementing undo/redo functionality, or evaluating expressions.
- **Queues:** FIFO (First-In-First-Out) principle, like a waiting line. You add elements to the back and remove them from the front. Queues are useful for processing tasks in a specific order, managing buffers, or simulating real-world queues.

Non-Linear Data Structures:

- **Trees:** Hierarchical structures with a root node, child nodes, and so on. Data is organized like a family tree. Trees enable efficient searching (especially binary search trees) and are used for sorting algorithms, representing file systems, or implementing decision trees in machine learning.
- **Graphs:** A collection of nodes (vertices) connected by edges. Useful for representing networks (social media, transportation), modeling relationships, or pathfinding algorithms.

Q.1.b) What do you understand by complexity of algorithm?

Answer:- Algorithm complexity refers to how efficiently an algorithm utilizes resources like time and space to solve a problem. It's like measuring how much work an algorithm needs to do as the size of the input data increases.

There are two main ways to analyze complexity:

- **Time Complexity:** This focuses on how long it takes the algorithm to run, typically measured in terms of basic operations executed. Imagine a simple algorithm that adds a list of numbers. As the list gets bigger (more input data), the number of addition operations increases. Time complexity is often expressed using Big O notation (O(n)), which represents the growth rate of operations based on the input size (n). Common complexities include:
    - Constant (O(1)): Operations don't depend on input size (e.g., accessing a specific array element).
    - Linear (O(n)): Operations grow proportionally to input size (e.g., iterating through a list).
    - Quadratic (O(n^2)): Operations grow quadratically with input size (e.g., nested loops comparing all pairs of elements).
    - Exponential (O(2^n)): Operations grow very rapidly with input size (usually undesirable due to slowness for large inputs).
- **Space Complexity:** This analyzes how much memory the algorithm uses during execution. It considers both the space for the input data itself and any additional data structures the algorithm creates. Ideally, space complexity should also grow modestly with the input size.

Q.2) What is linked list? Explain different types of linked list.

Answer :-

Linked List: A Dynamic Data Structure

Instead, each node contains data and a reference (or pointer) to the next node in the sequence, forming a chain-like structure. This allows for dynamic memory allocation, meaning the size of the list can grow or shrink as needed during program execution.

Structure of a Node

A typical node in a linked list has two parts:

- **Data:** This field stores the actual value or information associated with the node. The data type can vary depending on the application's needs (integers, strings, objects, etc.).
- **Next Pointer:** This field holds the memory address (reference) of the next node in the list. A special value, often null or NULL, indicates the end of the list.

Types of Linked Lists

1. **Singly Linked List:**
    - The most basic type of linked list.
    - Each node contains data and a pointer to the next node.
    - Traversal can only be done in one direction, starting from the head (the first node) and following the next pointers until reaching the null terminator.
2. **Doubly Linked List:**
    - Each node contains data, a pointer to the next node, and a pointer to the previous node.
    - Allows for traversal in both directions (forward and backward).
    - Useful when needing to insert or delete nodes frequently, as you can navigate the list from either end.
3. **Circular Linked List:**
    - A variation of the singly linked list.
    - The last node's next pointer points back to the head node, creating a circular chain.
    - Useful for applications where a continuous flow is needed, such as implementing a round-robin scheduler.

Choosing the Right Linked List

The choice of linked list type depends on the specific requirements of your program:

- **Singly Linked List:** Suitable for basic linear data storage when insertion/deletion at the beginning or end is frequent.
- **Doubly Linked List:** Preferred when two-way traversal is required, or when frequent insertion/deletion needs to occur anywhere in the list.
- **Circular Linked List:** Ideal for simulating a circular structure or when the end of the list needs to seamlessly connect back to the beginning.

Advantages of Linked Lists:

- **Dynamic Memory Allocation:** Nodes can be added or removed as needed, making them efficient for data sets of variable size.
- **Efficient Insertion/Deletion:** Insertion and deletion of nodes can be done at any position in the list with relative ease, especially compared to arrays.

Disadvantages of Linked Lists:

- **No Random Access:** Unlike arrays, you cannot directly access any element by its index in constant time. You need to traverse the list from the beginning until you reach the desired node.
- **Memory Overhead:** Each node stores an extra pointer reference, which can consume additional memory compared to arrays, especially for simple data types.

Q.3.a) Discuss PUSH and POP operations of STACK in detail.

Answer:- PUSH and POP: The Lifeblood of Stacks

A stack, following the LIFO (Last In, First Out) principle, operates like a stack of plates. You add plates (data) on top (push), and remove them from the top (pop). Two fundamental operations govern this behavior: PUSH and POP.

PUSH

- **Function:** Adds a new element (data) to the top of the stack.
- **Implementation:**
    1. Check if the stack is full (implementation-dependent). If full, handle overflow condition (e.g., error message).
    2. Create a new node to hold the data to be pushed.
    3. Update the next pointer of the new node to point to the current top node (or null if empty).
    4. Update the stack's top pointer to point to the newly created node. This becomes the new top of the stack.

POP

- **Function:** Removes and returns the element from the top of the stack.
- **Implementation:**
    1. Check if the stack is empty (underflow condition). If empty, return an error or special value.

2. Store the data from the top node (the one to be removed).
3. Update the stack's top pointer to point to the second node (the new top).
4. Decrement the stack size (if size is tracked).
5. Return the stored data (the element that was removed).

Key Points:

- PUSH and POP operations modify the top pointer of the stack, keeping track of the current top element.
- PUSH adds an element, while POP removes and returns it.
- Both operations require checks for overflow (PUSH) and underflow (POP) conditions to ensure proper stack behavior.

Applications of PUSH and POP:

- Function call handling (storing return addresses)
- Expression evaluation (postfix notation)
- Backtracking algorithms (undoing operations)
- Implementing undo/redo functionality in applications

Q.3.b) What is AVL tree? How it is different from BST?

Answer:- AVL Trees: Maintaining Balance in the BST World

An AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing type of Binary Search Tree (BST). Both BSTs and AVL trees are used for efficient searching and sorting, but AVL trees offer a guaranteed balance, ensuring faster operations.

Key Differences:

1. **Balance Factor:**
    - **BST:** No concept of balance factor. A BST can become unbalanced (heavily skewed to one side), leading to slower searches.
    - **AVL Tree:** Every node has a balance factor, which is the difference in heights between its left and right subtrees. AVL trees strictly maintain a balance factor of -1, 0, or 1 for each node, ensuring a balanced structure.
2. **Structure:**
    - **BST:** Simpler structure with only data and pointers to left and right child nodes.
    - **AVL Tree:** Requires an additional field in each node to store the balance factor.

3. **Operations:**
    - **BST:** Insertion and deletion are relatively straightforward, but performance can degrade with unbalanced trees.
    - **AVL Tree:** Insertion and deletion involve rotations to maintain balance. These rotations add complexity but guarantee faster search and retrieval times.

Choosing Between BST and AVL Tree:

- **BST:** Simpler to implement, suitable for situations where occasional imbalances are acceptable and search performance is not critical.
- **AVL Tree:** More complex but ensures faster search and retrieval due to its guaranteed balance. Ideal for applications where frequent insertions, deletions, and efficient searches are essential.
- BST prioritizes simplicity in structure and operations.
- AVL Tree prioritizes balance and guaranteed search performance at the cost of slightly more complex rotations during insertions and deletions.

SET - II

Q.4) Explain difference between linear and binary search with example.

Answer:- **Linear Search vs. Binary Search: Finding Your Way Through Data**

When searching for data in a list or array, you have two main options: linear search and binary search. While both aim to locate a specific element, they differ in their approach and efficiency. Here's a breakdown to help you choose the right tool for the job.

Linear Search: A Simple Walk-Through

Imagine searching for a book in a bookshelf that isn't organized. Linear search mimics this process. It starts at the beginning of the list and compares each element with the target value one by one, moving linearly until it either finds a match or reaches the end of the list.

Example:

Let's say you have a list of friends' names: ["Alice", "Bob", "Charlie", "David", "Emily"]. You want to find "Charlie".

- Linear search starts with "Alice". It compares "Alice" with "Charlie" and finds no match.
- It moves on to "Bob", again no match.
- This continues until it reaches "Charlie" and finds a match.

Pros of Linear Search:

- **Simple to understand and implement:** Linear search is straightforward and doesn't require any complex logic or data organization.
- **Works on any data:** It can be used on unsorted or partially sorted lists, unlike binary search.

Cons of Linear Search:

- **Slow for large datasets:** As the list size increases, the search time grows proportionally. In the worst-case scenario (target element is at the end), it needs to compare with every element.
- **Inefficient for frequent searches:** If you need to search for elements repeatedly in a large list, linear search becomes cumbersome.

Binary Search: Divide and Conquer

Binary search, on the other hand, takes a more strategic approach. It works best with sorted lists. Here's how it goes:

1. **Start at the middle:** It begins by identifying the middle element of the list.

2. **Compare with the target:** It compares the middle element with the target value you're searching for.
    - **Match:** If they are equal, you've found your element!
    - **Target is less:** If the target value is less than the middle element, the search continues in the **left half** of the remaining list.
    - **Target is greater:** If the target value is greater than the middle element, the search continues in the **right half** of the remaining list.
3. **Repeat steps 1 and 2:** This process of dividing the search space in half and focusing on the relevant half continues until the target is found or the search space becomes empty.

Example:

Consider the same list of friends' names, but now sorted alphabetically: ["Alice", "Bob", "Charlie", "David", "Emily"]. You want to find "Charlie".

- Binary search starts with the middle element, which is "Charlie" in this case. Perfect, you've found it in just one step!

Pros of Binary Search:

- **Faster for large datasets:** Binary search has a logarithmic time complexity (O(log n)), meaning the search time grows much slower than the list size. This makes it significantly faster for large sorted lists.
- **Efficient for repeated searches:** With sorted data, binary search is ideal for scenarios where you need to find elements repeatedly.

Cons of Binary Search:

- **Requires sorted data:** Binary search relies on the list being sorted in ascending or descending order. It won't work efficiently on unsorted data.
- **Slightly more complex to implement:** While the basic concept is understandable, implementing binary search requires more complex logic compared to linear search.

Q.5) What is external sorting? Explain in detail.

Answer:- **External Sorting: Taming Massive Datasets**

When dealing with datasets that are too large to fit comfortably in your computer's main memory (RAM), traditional sorting algorithms like quicksort or merge sort become impractical. This is where external sorting comes into play.

What is External Sorting?

External sorting is a class of sorting algorithms designed to handle massive amounts of data that reside on slower external storage devices like hard drives. These algorithms leverage a combination of internal memory (RAM) and external storage to efficiently sort the data in a step-by-step process.

Why External Sorting?

Imagine sorting a million phone numbers. In-memory sorting algorithms would struggle to hold everything in RAM, leading to slow performance due to frequent data swapping between RAM and the disk. External sorting breaks down the problem into manageable chunks, sorts them efficiently using internal memory, and then merges the sorted chunks back together on the disk.

How Does External Sorting Work?

There are two main approaches to external sorting:

1. **External Merge Sort:** This is a popular and efficient technique. Here's the general process:
    - **Pass 1: Divide and Conquer:** The data is split into smaller sub-lists that can fit in RAM. Each sub-list is then sorted using a regular in-memory sorting algorithm like quicksort or merge sort. These sorted sub-lists are called runs.
    - **Pass 2 (and Subsequent Passes): Merge Runs:** The sorted runs are merged pairwise or in small groups. This merging process happens in multiple passes. In each pass, a set of runs are read from the disk, merged in sorted order using internal memory, and written back to the disk as a new, larger sorted run. With each pass, the number of runs reduces, and their size increases.
    - **Final Pass:** This pass merges the remaining runs into a single, final sorted file.
2. **Distribution Sort:** This approach resembles quicksort but uses external storage. The data elements are partitioned based on their keys (values used for sorting) into multiple

buckets on the disk. Then, each bucket is sorted independently using an in-memory sorting algorithm. Finally, the sorted buckets are merged to form the final sorted output.

Benefits of External Sorting:

- **Handles Large Datasets:** Efficiently sorts data exceeding main memory capacity.
- **Scalability:** Can handle datasets that grow in size over time.
- **Flexibility:** Works with various data types and sorting algorithms.

Challenges of External Sorting:

- **Increased Complexity:** Requires more complex algorithms and logic compared to in-memory sorting.
- **Disk I/O Overhead:** Frequent data transfers between RAM and disk can impact performance if not optimized.
- **Temporary Storage:** May require additional storage space for temporary files during the sorting process.

Choosing the Right External Sorting Algorithm:

The choice between external merge sort and distribution sort depends on factors like:

- **Data Distribution:** If the data has a predictable or skewed distribution, distribution sort might be more efficient.
- **Available Memory:** If RAM is limited, external merge sort might be preferable due to its lower in-memory requirements during the merge phase.

Q.6) Explain collision resolution methods in detail.

Answer:- Hash tables are a fundamental data structure that rely on a hash function to map keys to bucket (or slot) locations in an array. However, collisions occur when multiple keys map to the same bucket. To maintain efficient search, insertion, and deletion operations, we need strategies to resolve these collisions.

1. Separate Chaining

- **Concept:** This method treats each bucket like the head of a linked list. When a collision occurs (multiple keys map to the same bucket), the new key-value pair is added as a new node at the end of the existing linked list associated with that bucket.
- **Implementation:**
    - Each bucket in the hash table stores a pointer to the head of a linked list (or null if empty).

- During insertion, if a collision occurs, a new node is created to hold the key-value pair, and it's linked to the end of the existing linked list.
- Searching involves traversing the linked list for the desired key.
- Deletion requires finding the node within the linked list and removing it.

- **Advantages:**
  - **Simple to implement:** Easy to understand and code.
  - **Flexible:** Can handle any number of colliding elements.
  - **Good for non-uniform data distribution:** Works well even if the hash function doesn't distribute keys uniformly across buckets.
- **Disadvantages:**
  - **Performance overhead:** Traversing linked lists during search and deletion can be slower than direct array access.
  - **Memory usage:** Requires additional memory for the linked list nodes.

2. Open Addressing

- **Concept:** This method attempts to find a new empty slot within the hash table itself for the colliding key-value pair. It uses a probing function to systematically search for an alternative slot.
- **Types of Probing:**
  - **Linear Probing:** The simplest approach. It checks the next slot (wrapping around if needed) until an empty slot is found.
  - **Quadratic Probing:** Uses a quadratic function (e.g., i^2) to determine the probe distance, aiming to spread out probes more evenly.
  - **Double Hashing:** Employs a secondary hash function to calculate the probe distance for each collision. This helps avoid clustering of elements.
- **Implementation:**
  - During insertion, if a collision occurs, the probing function is used to find an empty slot within the hash table.
  - Searching involves using the same probing function to traverse the table until the desired key is found or an empty slot is encountered (indicating the key is not present).
- **Advantages:**
  - **Faster search and deletion:** Direct array access can be quicker compared to linked list traversal.

- **Potentially less memory usage:** No separate linked list structures are needed.

- **Disadvantages:**
  - **Clustering:** If the hash function or probing function isn't well-designed, collisions can cluster together, leading to slowdowns.
  - **Primary clustering:** If a large number of keys map to the same initial bucket, finding an empty slot can become time-consuming.