Unit -1

DD MM YYYY

Basics of Data structures

Que- Explain the difference between data type, abstract data type and data structure.

⇒ The difference between data type, abstract data type and data structure :-

1] Data Type :-

a] A data type is a classification or categorization that specifies which type of data a variable can hold.

b] It defines the set of values that a variable can take on and the operations that can be performed on those values.

c] Common data types include integers, floating-point numbers, characters and booleans.

d] Data types are typically supported directly by programming languages and determine how data is stored in memory.

2] Abstract Data Type (ADT) :-

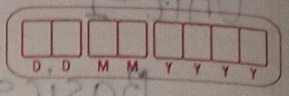
a] An abstract data type is a high-level description of a data structure that focuses on the operation that can be performed on the data and not on how they are implemented.

b] ADT's define a set of operations and their behavior but do not specify how these operation are implemented in code.

c] Examples of ADT's include stacks, queues, lists and dictionaries.

d] ADT's provide a blueprint for creating data structures but leave the actual implementation details to the programmer.

Types of ADT :- 1] List
2] stack
3] queue



a] Data structure :-
a] A data structure is concrete implementation of an abstract data type (ADT)
b] It involves organizing and storing data in a specific way in memory to support efficient operations.
c] Data structures determine how data is stored, accessed and manipulated.
d] Examples of data structures include arrays, linked Lists, trees, graphs and hash tables.
e] Data structures provide the low-level details of how data is managed in a program.

Que- Explain the need of a data structure in programming point of view.

⇒

Need of a Data structure in programming Point of view :-

1] Data organization :-
Data ~~organiz~~ structures provide a way to efficiently organize and store data. This organization is essential for quick and easy access to information within a program.

2] Memory management :-
Data structures allow for efficient memory allocation and management. They help ensure that memory is used effectively and can be released when it's no longer

needed, preventing memory leaks.

3] Abstraction :-
Data structures abstract the underlying implementation details.
- programmers can work with high-level data structures like lists or sets without needing to understand the low-level details of how data is stored and retrieved.

4] Complex problem solving :-
Many programming problems involve complex data relationships and manipulations.
- Data structures like trees and graphs are essential for solving these kinds of problems efficiently.

5] Code Reusability :-
- Using established data structures allows for code reusability.
- Many programming languages provide built-in data structures, and there are libraries that offer specialized data structures for various purposes.
- This saves time and effort in development.

6] Algorithm Design :-
Data structures often go hand in hand with algorithms.
- Efficient algorithms are designed with specific data structures in mind.

- The choice of data structure can profoundly impact the efficiency of an algorithm.

7) Scalability :-
Data structures help in building scalable solutions.
- As data volumes grow, the choice of the right data structure becomes critical for maintaining the performance of the program.

8) Debugging and Maintenance :-
- Well-designed data structures make code easier to debug and maintain.
- clear data structures make it easier to understand the code's logic and find and fix issues.

Que- What is Data structure, List the Basic operations of Data structures.

→ Data structure :-
Data structure are fundamental constructs used to organize and manage data in computer programs.

List of Basic operations :-

1) Arrays :-
a) Access (Read/write) :- Retrieve or modify an element at a specific index.

b) insertion :- Add an element at the end or a specified index.
c) Deletion :- Remove an element from a specific index.
d) Search :- Find the index of a specific element.

2) Linked Lists :-
a) Insertion :- Add a new node at the beginning end or a specific position.
b) peletion :- Remove a node from a specific position.
c) search :- Find a node with a specific value.

3) Stacks :-
a) Push :- Add an element to the top of the stack.
b) Pop :- Remove and return the top element.
c) Peek :- View the top element without removing it.

4) Queues :-
a) Enqueue :- Add an element to the back of the queue.
b) Dequeue :- Remove and return the front element.
c) Peek :- view the front element without removing it.
check if the queue is empty.

5] Trees :-
a] Insertion :- Add a new node.
b] Traversal :- Visit nodes in specific order.
c] search :- Find a node with a specific value.
d] Deletion :- Remove a node.

6] Graphs :-
a] Insertion :- Add a vertex or edge
b] Deletion :- Remove a vertex or edge
c] Traversal :- Visit vertices and edges.

7] Hash Tables :-
a] Insertion :- Add a key-Value pair
b] Lookup :- Retrieve the value associated with a key.
c] Deletion :- Remove a key-Value pair.
d] Hashing :- convert a key into an index for efficient retrieval.

8] Heaps :-
a] Insertion :- Add an element while maintaining the heap property.
b] Extraction :- Remove and return the element with the highest or lowest priority.
c] Peek :- view the highest or lowest Priority element.

9] sets and maps :-
a] Insertion :- Add an element with or without associated data.

b) Deletion :- Remove an element by key (map)
c) Lookup :- Retrieve an element by key (map)

Que- State the merits and demerits of linked list over array.

⇒ Merits of Linked Lists over Arrays :-

1) Dynamic size :-
- Linked lists can dynamically grow or shrink in size.
- This flexibility allows for efficient memory usage as you only allocate memory for the elements you need, unlike arrays, which have a fixed size.

2) Efficient Insertions and Deletions :-
- Insertions and deletions are generally more efficient in linked lists, especially when dealing with elements in the middle of List.

3) No wasted Memory :-
- Linked lists don't suffer from memory wastage due to unused slots, as in the case of arrays with preallocated sizes.

4) No Need for copying :-
Adding or removing elements from a linked list typically doesn't involve copying the entire data structure, as it might in arrays when resizing.

5] Support for complex Data structures:-
Linked lists are fundamental in building more complex data structures like stacks, queues and graphs.

- Demerits of Linked Lists over Arrays:-

1] Random Access:-
- Linked lists provide slow random access.
- To access an element at a specific position, you often have to tranverse the list from the beginning, which takes O(n) time in the worst case.

2] Memory overhead:-
- Each element in a linked list requires additional memory for the pointer/reference to the next element, resulting in higher memory overhead compared to arrays.

3] Cache Inefficiency:-
- Linked lists can be less cache-friendly.
- since elements are scattered in memory, accessing them may result in more cache misses compared to the contiguous memory layout of arrays.

4] Complexity:-
- Linked lists introduce more complexity due to the need to manage pointers/references, making them potentially

error-prone, especially in terms of memory management.

5] Additional pointer operations :-
- Traversing and manipulating linked lists require extra pointer operations, which can make code less readable and more challenging to debug.

6] Fixed storage complexity :-
Some types of linked lists don't support reverse traversal efficiently.
- To traverse in reverse, you may need to create a new linked list or use additional data structures, which can be inefficient.

Que- Explain Time complexity and space complexity with suitable example.

⇒

Time complexity :-
- Time complexity measures the amount of time an algorithm takes to complete its execution as a function of the input size.
- It provides an estimate of how the algorithm's runtime grows with increasing input size.
- It is usually expressed using Big O notation such as $O(n)$, $O(n^2)$, etc.

Example :- Linear Search
Suppose you have an array of n elements.

and you want to find a specific element in array.
- In the worst-case scenario, where the element is at the end of the array or not present, you would have to check each element one by one.
- The time complexity of a linear search is O(n), as the number of comparisons is directly proportional to the size of the input array (n).

2) space complexity :-
- space complexity measures the amount of memory space an algorithm uses as a function of the input size.
- It helps determine how much memory is required to execute an algorithm efficiently.
- Example :- Merge sort
- Merge sort is a sorting algorithm that uses a divide-and-conquer approach.
- It divides the input array into two halves, recursively sorts each half, and then merges them back together.
- In terms of space complexity, merge sort requires additional memory to store temporary arrays during the merge step.
- The space complexity of merge sort is O(n), where n is the size of the input array, because it needs to create temporary arrays of the same size as the input.

que- Difference between arrays and linked lists

⇒

Array	Linked list
1] Arrays are stored in contiguous location	1] Linked lists are not stored in contiguous location.
2] Fixed in size	2] Dynamic in size
3] Memory is allocated at compile time	3] Memory is allocated at run time.
4] Uses less memory than linked lists	4] Uses more memory because it stores both data and the address of next node.
5] Elements can be accessed easily.	5] Element accessing requires the traversal of whole linked list.
6] Insertion and deletion operation takes time	6] Insertion and deletion operation is faster.

que- Explain the various data representation in detail with suitable examples.

⇒

Various Data Representation :-

1] Arrays :-

- An array is a contiguous block of memory where elements are stored at consecutive memory locations.
- Each element in an array is accessed

using an index.
- Example:- An array of integers 'arr' where 'arr[0]=5', 'arr[1]=10' and 'arr[2]=15'.

2] Linked Lists:-
- A linked list is a data structure where each element (node) contains both data and a reference (pointer) to the next node in the sequence.
- Example:- A singly linked list with elements 10 -> 20 -> 30, where each node contains a value and a reference to the next node.

3] Stacks:-
- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) Principle.
- Elements are added and removed from one end (the top).
- Example:- A stack of integers where you push elements onto the top and pop element from the top.

4) Queues:-
- A queue is a linear data structure that follows the First-In-First-Out (FIFO) Principle.
- Elements are added at one end (rear) and removed from the other end (front).

• Example :- A queue of people waiting in line; where people are added at the end of the line and served from the front.

5) Trees :-
• Trees are hierarchical data structures consisting of nodes with parent-child relationships.
• Example :- A binary search tree (BST) where each node has a value, a left child (with a smaller value) and a right child (with a larger value).

6) Graphs :-
• Graphs represent a collection of nodes (vertices) and edges that connect these nodes.
• Example :- A social network graph where users are nodes, and friendships are edges connecting them.

7) Hash Tables :-
• Hash tables use a hash function to map keys to indices in an array, allowing for efficient key-value pair storage and retrieval.
• Example :- A dictionary where word (keys) are mapped to their definitions (values) using hash table.

Que:- Define, What is Algorithm and explain the different ways of analysing algorithm.

⇒

Algorithm :-
- An algorithm is a step-by-step procedure or a set of well-defined rules for solving a problem or accomplishing a specific task.

Different ways of analysing algorithm :-

1] Time complexity Analysis :-
- Time complexity measures how the running time of an algorithm grows as a function of the input size. It helps determine how efficient an algorithm.
- Common notations used for time complexity analysis include Big O (O), Big omega (Ω) and Big Theta (Θ).
- Example :- An algorithm with a time complexity of $O(n^2)$ means that its running time increase quadratically with the input size.

2) space Complexity Analysis :-
- space complexity measures how much memory an algorithm consumes in relation to the input size. It helps assess the algorithm's memory efficiency.
- Like time complexity, space complexity is also analyzed using Big O notation.
- Example :- An algorithm with a space

Complexity of O(n) indicates that it uses linear memory in relation to the input size.

3] Best case, worst case and Average case :-
Analysis :-

- Algorithms may perform differently under different scenarios. Analyzing best-case, worst-case and average-case scenarios helps understand their behavior in various situations.
- Best-case analysis identifies the minimum possible runtime.
- Worst-case analysis identifies the maximum possible runtime.
- Average-case analysis provides a more realistic expectation of performance based on probabilities.

4] Asymptotic Analysis :-

- Asymptotic analysis focuses on how an algorithm behaves as the input size approaches infinity. It helps identify the dominant factors affecting the algorithm's performance.
- It often involves simplifying the time or space complexity expression to its most significant term.
- Example :- Identifying that an algorithm's time complexity is $O(n^2)$ allows us to know that its efficiency decreases as the input size grows.

5. Empirical Analysis :-
- Empirical analysis involves running an algorithm on real input data and measuring its actual performance.
- It is useful for practical assessments but may not provide a precise mathematical description of an algorithm's behavior.

6. Amortized Analysis :-
- Amortized analysis considers the average time or space usage per operation over a sequence of operations.
- It helps determine whether an algorithm's occasional costly operations are balanced by cheaper operations, resulting in good overall performance.

Que- Explain Recursion and types of recursion with a C program to find the factorial of N given numbers.

⟹

Recursion :-
Recursion is a programming technique where a function calls itself to solve a problem.

Types of recursion :-

1. Direct Recursion :-
In this type of recursion, a function calls itself directly.

2] Indirect Recursion :-

In this type of Recursion, two or more functions call each other in a circular manner, eventually leading to a base case.

C program to Find the factorial of a given number :-

```
#include <stdio.h>
int factorial (int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial (n-1);
    }
}
int main()
{
    int n;
    Printf("Enter a non-negative integer : ");
    scanf("%d", &n);
    if(n<0)
    {
        Printf("Factorial is undefined");
    }
    else
    {
```

```
    int result = factorial(n);
    printf("Factorial of %d is %d\n", n,
    result);
}
    return 0;
}
```

Que- Illustrate the recursive function with help of Tower of Honai.

⇒

- The Tower of Hanoi is a classic problem that is often used to illustrate recursion.
- It involves three rods and a number of disks of different sizes, which can slide onto any rod.
- The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top.
- The goal is to move the entire stack to another rod, subject to the following rules:-

1] only one disk can be moved at a time.
2] A disk can only be placed on top of a larger disk or an empty rod.
3] The objective is to move all the discs to the forget rod, using the spare rod as an intermediary, while following the above rules.

Recursive Solution:-

The Tower of Hanoi problem can be elegantly solved using recursion.

- Here's how the recursive solution works

step by step:-

1) Base case:-
When there is only one disk (n=1) on the source peg, you can move it directly to the destination peg. This is the simplest case and serves as the base case for the recursion.

2) Recursive step:-
- To move n disks from the source peg to the destination peg, you can break it down into three subproblems:-
  1) Move the top n-1 disks from the source peg to the auxiliary peg
  2) Move the largest nth disk from the source peg to the destination peg.
  3) Move the n-1 disks from the auxiliary peg to the destination peg (using the source peg as auxiliary).

```
void Hanoi(int n, string a, string b, string c)
{
if(n==1) /* Base case */
  move(a,b);
else
{
Hanoi(n-1, a, c, b);
move(a,b);
Hanoi(n-1, c, b, a);
}
```

Que- Explain the classification Data structure.

⇒

Classification of Data structure:-

1) Primitive OR non-Primitive:-

a) Primitive:- These are fundamental data structures provided by programming languages such as integers, floats, characters and booleans.

- They store a single data element.

b) NON-Primitive:- They are constructed using primitive data types to store collections of data elements.

- Examples include arrays, lists, stack

2) Static or Dynamic:-

a) Static:- These have a fixed size that is determined at the time of creation and cannot be changed during runtime.

- Arrays are an example of static data structure

b) Dynamic:- These can grow or shrink in size during program execution.

- Dynamic arrays, linked list and trees are example.

3) Linear OR Non-Linear:-

a) Linear:- Data elements are organized sequentially and each element has a unique Predecessor and successor.

- Example include arrays, linked lists, stack

b) Non-Linear :- Data elements are organized in a hierachical or arbitrary manner and elements may have multiple predecessors or successors.
- Examples include trees and graphs

4) Homogeneous or Heterogeneous :-
a) Homogeneous :- All element in the structure are of the same data type.
- For example an array of integers.

b) Heterogeneous :- Elements in the structure can be of different data types.
- Example, , a struct or record in some programming languages can contain different data types within a single structure.