B. TECH - CS
Assignment - 1
Semester-I (Odd), Session: 2023-24
BCS-301: DATA STRUCTURE

Unit-1 Unit-Name: Introduction, Arrays and Link Lists	Course Outcome: CO1 – To Describe how arrays, linked lists, stacks, queues, trees, and graphs are represented in memory, used by the algorithms and their common applications.
Date of Distribution:20/09/2023	Faculty Name: Dr. Ashish Avasthi

Sr.	**MANDATORY QUESTIONS**	BL
1	Discuss the various properties of the algorithm that it must contain. Ans: Properties of Algorithm Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm. Non Ambiguity Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in any algorithm should not denote any conflicting meaning. This property also indicates the effectiveness of algorithm. Range of Input The range of input should be specified. This is because normally the algorithm is input driven and if the range of input is not being specified then algorithm can go in an infinite state. Multiplicity The same algorithm can be represented into several different ways. That means we can write in simple English the sequence of instruction or we can write it in form of pseudo code. Similarly, for solving the same problem we can write several different algorithms. Speed The algorithm is written using some specified ideas. Bus such algorithm should be efficient and should produce the output with fast speed. Finiteness The algorithm should be finite. That means after performing required operations it should be terminate.	2
2	Write a program to input two m x n matrices and then calculate the sum of their corresponding elements and store it in a third m x n matrix.	4

```
Ans: #include <stdio.h>
int main()
{
   int a[3][3], b[3][3], result[3][3];

   // Taking input using nested for loop
   printf("Enter elements of 1st matrix\n");
   for (int i = 0; i < 3; ++i)
      for (int j = 0; j < 3; ++j)
      {
         scanf("%d", &a[i][j]);
      }

   // Taking input using nested for loop
   printf("Enter elements of 2nd matrix\n");
   for (int i = 0; i < 3; ++i)
      for (int j = 0; j < 3; ++j)
      {
         scanf("%d", &b[i][j]);
      }

   // adding corresponding elements of two matrices
   for (int i = 0; i < 3; ++i)
      for (int j = 0; j < 3; ++j)
      {
         result[i][j] = a[i][j] + b[i][j];
      }

   // Displaying the sum
   printf("Sum Of Matrix:\n");

   for (int i = 0; i < 3; ++i)
   {
      for (int j = 0; j < 3; ++j)
      {
         printf("%d\t", result[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

3	Why is a doubly linked list more useful than a singly linked list? Ans:	2

Singly Linked List	Doubly Linked List
A Singly Linked has nodes with a data field and a next link field.	A Doubly Linked List has a previous link field along with a data field and a next link field.
In a Singly Linked List, the traversal can only be done using the link of the next node.	In a Doubly Linked List, the traversal can be done using the next node link as well as the previous node link.
A Singly Linked List occupies less memory than the Doubly Linked List, as it has only 2 fields.	A Doubly Linked List occupies more memory than the Singly Linked List, as it has 3 fields.
Accessing elements in a Singly Linked List is less efficient when compared to a Doubly Linked List, as only forward traversal is possible.	Accessing elements in a Doubly Linked List is more efficient whe compared to a Singly Linked Lis both forward and backward traversal is possible.
The time complexity of inserting or deleting a node at a given position (if the pointer to that position is given) in a singly linked list is O(n).	The time complexity of inserting deleting a node at a given positic (if the pointer to that position is given) in a doubly linked list is O(1).

	A single linked list is preferred when we have memory limitation (we can't use much memory) and searching is not required.	A doubly linked list is preferred when we don't have memory limitations and searching is required (we need to perform a search operation on the linked li	
4	What do you understand by the term 'Worst Case' in context to an Algorithm? Ans: In computer science (specifically computational complexity theory), the worst-case complexity measures the resources (e.g. running time, memory) that an algorithm requires given an input of arbitrary size (commonly denoted as n in asymptotic notation). It gives an upper bound on the resources required by the algorithm. In the case of running time, the worst-case time complexity indicates the longest running time performed by an algorithm given any input of size n and thus guarantees that the algorithm will finish in the indicated period. The order of growth (e.g., linear, logarithmic) of the worst-case complexity is commonly used to compare the efficiency of two algorithms. The worst-case complexity of an algorithm should be contrasted with its average-case complexity, which is an average measure of the number of resources the algorithm uses on a random input.		2
5	With the help of a program show how to pass an array to a function. Ans: In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values. As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter. Consider the following syntax to pass an array to the function. functionname(arrayname);//passing array Methods to declare a function that receives an array as an argument		4

There are 3 ways to declare the function which is intended to receive an array as an argument.
First way:
return_type function(type arrayname[])
Declaring blank subscript notation [] is the widely used technique.
Second way:
return_type function(type arrayname[SIZE])
Optionally, we can define size in subscript notation [].
Third way:
return_type function(type *arrayname)

6 Write a program in C language for merging of two arrays. 3

Ans:
```
#include <stdio.h>
int mergearray(int a[], int b[], int arr1size, int arr2size)
{

   // resultant Array Size Declaration
   int arr_resultsize = arr1size + arr2size;
   int c[arr_resultsize], i, j;
    // copying array 1 elements in to c array
   for (i = 0; i < arr1size; i++) {
      c[i] = a[i];
   }
    // copying array 2 elements in to c array
   for (i = 0, j = arr1size;
       j < arr_resultsize && i < arr2size; i++, j++) {
      c[j] = b[i];
   }
    // Array Elements After Merging
   for ( int k = 0; k < arr_resultsize; k++) {
      printf("%d ", c[k]);
   }
}
 int main()
{
   int arr1size = 5, arr2size = 5;

   // elements of first Array
   int a[5] = { 1, 2, 3, 4, 5 };

   // elements of Second Array
   int b[5] = { 6, 7, 8, 9, 10 };

   printf("%d", mergearray(a, b, arr1size, arr2size));
```

	`return 0;`
	`}`	

7	Explain the complexity of algorithms and time-space trade-off. Ans: The trade-offs between time and space complexity are often inversely proportional, meaning that improving one may worsen the other. For example, a hash table is a data structure that can perform insertions, deletions, and searches in constant time, O(1), regardless of the input size. However, a hash table also requires a lot of memory to store the data and avoid collisions, which are situations where two different keys map to the same index in the table. A linked list, on the other hand, is a data structure that can perform insertions and deletions in constant time, O(1), but requires linear time, O(n), to search for an element. A linked list also requires less memory than a hash table, as it only stores the data and a pointer to the next node.	2

8	What is array? How it is differ from pointer? Explain with suitable example.
	Ans: An array is an arrangement of numbers, pictures or objects formatted into rows and columns according to their type. In coding and programming, an array is a collection of items, or data, stored in contiguous memory locations, also known as database systems . The main difference between Array and Pointers is the fixed size of the memory block. When Arrays are created the fixed size of the memory block is allocated. But with Pointers the memory is dynamically allocated.
	`#include <stdio.h>`
	
	`int main()`
	`{`
	`int arr[] = {10, 20, 30, 40, 50, 60};`
	`int *ptr = arr;`
	`printf("arr[2] = %d\n", arr[2]);`
	`printf("*(arr + 2) = %d\n", *(arr + 2));`
	`printf("ptr[2] = %d\n", ptr[2]);`
	`printf("*(ptr + 2) = %d\n", *(ptr + 2));`
	`return 0;`
	`}`
		2

9	Define data structure. Explain primitive and non-primitive data structures in details. Ans: A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.	2

A primitive data type is the most basic kind of data structure used in programming languages. Primitives are the building blocks for more complex structures, and the most commonly used primitive type is the int. An int is an integer, which is an entire number (positive, negative or zero) without decimals.

The difference between primitive and non primitive data types is that nonprimitives can store multiple values within one object, while primitives only store single values. For example, a nonprimitive such as an array can contain several integers (like 10, 50 and 25), whereas a primitive like int can only store one integer at one time (such as 10).

Primitives are useful in programming because they allow us to create customized methods that manipulate different kinds of data when creating software applications. By default, most programming languages have set methods that are used to handle primitive types. When you are writing code, you have to specify which data type you want to use when declaring variables (i.e., int x = 3).

It's helpful to understand the difference between primitive and non-primitive types when coding since it will help you write better programs and make your code easier to read and maintain over time. Understanding primitives will also help you become a better programmer by allowing you to write more efficient programs that perform specific tasks faster while also reducing memory usage.

When it comes to programming, it is important to understand the difference between primitive and non-primitive data types. Primitive data type, also known as atomic or primary data types, are the baselevel data structures used in programming languages. Examples of primitive types include integers, floats, booleans, and strings. Each primitive type is determined by its value range and operations that can be performed on them.

Nonprimitive data types on the other hand, consist of complex data structures that are derived from existing primitive types. Examples of difference between primitive and non-primitive types include Arrays and objects. These data types can store multiple values at once and provide a layer of abstraction from the underlying logic used to store those values. Arrays for example, allows you to store multiple values within a single variable so that you don't need to define individual variables each time you want to access an item in the array. Objects are more complex than arrays as they contain custom properties that define how the object behaves when operated upon.

It is important for any programmer to understand both difference between primitive and non-primitive types in order to efficiently write code that is intelligible and organized. By understanding the difference between primitive and non-primitive of data structures, your code will end up being

more efficient in terms of storage space, execution time and clarity for other developers reading your code.

10 Explain sparse matrix. 2

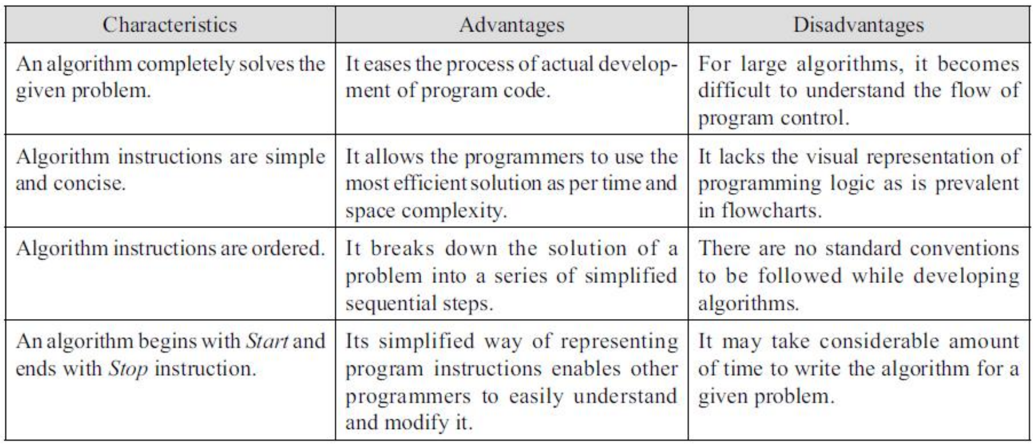
Ans: A matrix is a two-dimensional data object made of m rows and n columns, therefore having a total m x n value. If most of the elements of the matrix have a 0 value, then it is called a sparse matrix.

The number of zero-valued elements divided by the total number of elements (e.g., m × n for an m × n matrix) is called the sparsity of the matrix (which is equal to 1 minus the density of the matrix). Using those definitions, a matrix will be sparse when its sparsity is greater than 0.5.

A sparse matrix is common in most of the scenarios of natural language processing. Choosing the applicable compressing technique and efficiently utilizing memory and computational power is key while analyzing the unstructured data.

11 Discuss the characteristics, advantages and disadvantages of the algorithm. 2

Ans:



Characteristics	Advantages	Disadvantages
An algorithm completely solves the given problem.	It eases the process of actual development of program code.	For large algorithms, it becomes difficult to understand the flow of program control.
Algorithm instructions are simple and concise.	It allows the programmers to use the most efficient solution as per time and space complexity.	It lacks the visual representation of programming logic as is prevalent in flowcharts.
Algorithm instructions are ordered.	It breaks down the solution of a problem into a series of simplified sequential steps.	There are no standard conventions to be followed while developing algorithms.
An algorithm begins with *Start* and ends with *Stop* instruction.	Its simplified way of representing program instructions enables other programmers to easily understand and modify it.	It may take considerable amount of time to write the algorithm for a given problem.

12 Explain the following algorithm approaches: 3

A) Greedy Algorithm: Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So, the problems were choosing locally optimal also leads to global solution are the best fit for Greedy. For example, consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to a globally optimal solution because we are allowed to take fractions of an item.

B) Divide and Conqueror : Divide and conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy. A typical Divide and Conquer algorithm solve a problem using the following three steps:

Divide: This involves dividing the problem into smaller sub-problems.

Conquer: Solve sub-problems by calling recursively until solved.
Combine: Combine the sub-problems to get the final solution of the whole problem.
C) Backtracking: The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.
D) Randomized Algorithms: A randomized algorithm is a technique that uses a source of randomness as part of its logic. It is typically used to reduce either the running time, time complexity; or the memory used, or space complexity, in a standard algorithm.

13 Discuss the push operation on stack by using array. 2

Ans: // C program for array implementation of stack

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
// A structure to represent a stack
struct Stack {
        int top;
        unsigned capacity;
        int* array;
};
// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
        struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
        stack->capacity = capacity;
        stack->top = -1;
        stack->array = (int*)malloc(stack->capacity * sizeof(int));
        return stack;
}
// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
        return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
        return stack->top == -1;
}
```

```
// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
        if (isFull(stack))
                return;
        stack->array[++stack->top] = item;
        printf("%d pushed to stack\n", item);
}
// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
        if (isEmpty(stack))
                return INT_MIN;
        return stack->array[stack->top--];
}
// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
        if (isEmpty(stack))
                return INT_MIN;
        return stack->array[stack->top];
}
// Driver program to test above functions
int main()
{
        struct Stack* stack = createStack(100);
        push(stack, 10);
        push(stack, 20);
        push(stack, 30);
        printf("%d popped from stack\n", pop(stack));
        return 0;
}
```

14	What is Linked list. Discuss the algorithm for traversing the linked list. Ans: A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list. STEP 1: SET PTR = HEAD. STEP 2: IF PTR = NULL. STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL. STEP 5: PRINT PTR→ DATA. STEP 6: PTR = PTR → NEXT. STEP 7: EXIT.	2

15	Write a algorithm to search the data in the linked list. Ans: Step 1: SET PTR = HEAD. Step 2: Set I = 0. STEP 3: IF PTR = NULL. WRITE "EMPTY LIST" GOTO STEP 8. END OF IF. STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL. STEP 5:IF ptr → data = item. WRITE i+1. End of IF. STEP 6: I = I + 1. STEP 7:PTR = PTR → NEXT. [END OF LOOP] STEP 8: EXIT.	4
16	What are the limitation of Arrays of pointers to string? Ans: Array of pointers to strings is a widely used data structure in C programming. While it provides flexibility and efficient string management, it also has certain limitations that programmers need to be aware of. In this article, we will explore the limitations of the array of pointers to strings in C, and understand how to overcome them for effective programming. Limited Size and Fixed Length One of the primary limitations of an array of pointers to strings is its fixed size and length. Consider the following: Limited Size: The size of the array needs to be determined at compile-time. It may not be possible to accommodate a variable number of strings or dynamically adjust the size. Fixed Length: Strings within the array have fixed lengths, determined by the allocated memory. It becomes challenging to handle strings of different lengths or accommodate larger strings. Memory Fragmentation Another limitation of the array of pointers to strings is the potential for memory fragmentation. Let's explore this further: Non-Contiguous Memory: Each string in the array is stored in separate memory locations. This can lead to fragmented memory, with strings scattered across different memory areas. Inefficient Memory Utilization: Memory fragmentation can result in inefficient memory utilization. It may become challenging to allocate and deallocate memory efficiently for string storage.	4

	Lack of Dynamic Resizing The array of pointers to strings lacks the ability to dynamically resize itself. Consider the following: Inability to Resize: Once the array is declared and initialized, it cannot be resized automatically. Adding or removing strings from the array requires manual memory reallocation and string copying. Time and Space Complexity: Dynamic resizing involves additional time and space complexity. The programmer needs to carefully manage memory allocation and deallocation to avoid memory leaks or excessive overhead. Difficulty in Sorting and Searching Sorting and searching operations can be challenging with an array of pointers to strings. Let's discuss this limitation: Inefficient Sorting: Sorting the array of pointers to strings based on string content can be complex. Custom sorting algorithms or libraries may be required to handle such operations efficiently. Inefficient Searching: Searching for a particular string within the array can involve traversing multiple pointers. It may not be as efficient as other data structures like hash tables or binary search trees. Workarounds and Alternative Data Structures To overcome the limitations of the array of pointers to strings, consider the following workarounds and alternative data structures: Dynamic Memory Allocation: Use dynamic memory allocation techniques like malloc() and free() to handle variable-sized strings. Linked List: Implement a linked list of string nodes to allow dynamic resizing and efficient memory utilization. Hash Tables or Binary Search Trees: Utilize data structures like hash tables or binary search trees for efficient searching and sorting operations.	
	SUPPLEMENTARY QUESTIONS	
1	Write a program to interchange the biggest and the smallest number in an array.	4

2	Suppose U is the text 'MARC STUDIES MATHEMATICS'. Use INSERT to change U so that it reads: (A). MARC STUDIES ONLY MATHEMATICS. (B). MARC STUDIES MATHEMATICS AND PHYSICS (C). MARC STUDIES APPLIED MATHEMATICS.	4

REFERENCES

TEXT BOOKS:				
Ref. [ID]	**Authors**	**Book Title**	**Publisher/Press**	**Edition &Year of Publication**
[T1]	Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augensteini	Data Structures Using C and C++	PHI Learning Private Limited	5th 2016
[T2]	P. S. Deshpandey	C and Data structure	Wiley Dreamtech Publication	5th 2018
REFERENCE BOOKS:				
Ref. [ID]	**Authors**	**Book Title**	**Publisher/Press**	**Edition &Year of Publication**
[R1]	Reema Thareja	Data Structure Using C	Oxford Publication	2015

Signature of Faculty:_______________
(With Date)

Signature of HOD:________________
(With Date)