# DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

### I YEAR / I SEMESTER

# IMPORTANT QUESTIONS

## 23MC102 – Advanced Data Structure and Algorithms

1. **How does the Asymptonic notation Work?**

➔Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω).

➔Asymptotic analysis is a mathematical technique used for understanding the behavior of algorithms as their input increases.

➔ It uses asymptotic notations to describe the growth rate or time complexity of an algorithm, which allows us to compare different algorithms and understand how they perform in realistic scenarios.

2. **Define the term Dynamic datastructure?**

➔A dynamic data structure is a data structure that can change its size or structure during runtime, allowing it to grow or shrink as needed.

➔ This is in contrast to a static data structure, which has a fixed size and cannot change during runtime.

➔Dynamic data structures, such as linked lists, stacks, queues, trees, graphs, and hash tables, are those that have a variable size and structure at runtime.

➔This means they can be modified or resized as needed.

3. **What is Recursive algorithms?**

➔Recursive algorithms are a method of problem solving where the solution to a problem depends on solutions to smaller instances of the same problem.

➔They involve a function that calls itself during its execution. Unravelling and solving of the sub-problems is done by using the concept of recursion in programming.

HTML, CSS, JAVASCRIPT, REACT.

➔ Recursion works by creating a stack of function calls. When a function calls itself, a new instance of the function is created and pushed onto the stack.

➔ This process continues until a base case is reached, which is a condition that stops the recursion.

4. **Define Program? What are all Phases of Program Development Lifecycle?**

➔A program is an application program created through the process of program development. The program development life cycle (PDLC) is a sequence of steps used to develop a program in any programming language.

➔The phases of the PDLC are analyzing, designing, coding, debugging and testing, and implementing and maintaining application software.

5. **How is Best case different from Worst case of Complexities Algorithm?**

➔Best case is the function which performs the minimum number of steps on input data of n elements.

➔Worst case is the function which performs the maximum number of steps on input data of size n.

➔ complexity measures the resources (e.g. running time, memory) that an algorithm requires given an input of arbitrary size .

| Algorithm | Best case | Average case | Worst case |
|---|---|---|---|
| Quick sort | O(n log(n)) | O(n log(n)) | $O(n^2)$ |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Merge sort | O(n log(n)) | O(n log(n)) | O(n log(n)) |
| Bogo sort | O(n) | O(n n!) | O(∞) |

**6. What is an Array ? and its Operations with an example?**

➔ An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations.

➔Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.

➔ The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update.

➔. These operations are usually performed to either modify the data in the array or to report the status of the array. Following are the basic operations supported by an array.

➔For example, consider a program that needs to store information about a group of employees, including their names, salaries, and hire dates.

**7. Define the term "Sparse Matrices" and its Creation and Addition of Algorithm?**

➔ A sparse matrix is a special case of a matrix in which the number of zero elements is much higher than the number of non-zero elements..

➔ As a rule of thumb, if 2/3 of the total elements in a matrix are zeros, it can be called a sparse matrix

➔ A sparse matrix is one with a large number of zero entries.

➔ A more practical definition is that a matrix is sparse if the number or distribution of the zero entries makes it worthwhile to avoid storing or operating on the zero entries.

➔ Two elements with the same row values are further sorted according to their column values.

➔ Now to Add the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix.

**8. How do you Difference Between the Insertion Sort and Bubble Sort ?**

➔ Although direct insertion sort uses sequential search, when the insertion position is found, the element has already been moved.

➔ Bubble sort gets its name from the fact that data "bubbles" to the top of the dataset. Bubble sort is alternatively called "sinking sort" for the opposite reason, which is that some elements of data sink to the bottom of the dataset.

➔ In the bubble sort algorithm, we check the neighbour element and swap them if required.

➔ In the insertion sort, we transfer an element at one time to construct a sorted array.

**9. Examine the role of the Linked list ?And its Types?**

➔Data structures bring together the data elements in a logical way and facilitate the effective use, persistence and sharing of data.

➔They provide a formal model that describes the way the data elements are organized. Data structures are the building blocks for more sophisticated applications.

➔ Data structures are a specific way of organizing data in a specialized format on a computer.

➔so that the information can be organized, processed, stored, and retrieved quickly and effectively. They are a means of handling information, rendering the data for easy use.

**10. Differentiate between Linear Search and Binary Search in Data Structure?**

➔ Linear Search, ideal for smaller datasets, examines each element sequentially. In contrast, Binary Search efficiently navigates larger, sorted datasets by halving the search interval.

➔Choose wisely to optimize your algorithm's performance and ensure rapid, precise outcomes in your computing tasks.

➔ In a linear search, each element in the list is sequentially searched one after the other until it is found in the list.

➔A binary search finds the list's middle element recursively until the middle element matches a searched element.

**Part B – Question & Answer**

11 **Outline an Introduction to Data Structures? What is Need for Data Structures?**

➔ Data Structures is about how data can be stored in different structures. Algorithms is about how to solve different problems, often by searching through and manipulating data structures.

➔Theory about Data Structures and Algorithms (DSA) helps us to use large amounts of data to solve problems efficiently.

➔Data structures are essential for two main reasons: they make the code more efficient, and they make the code easier to understand.

➔When it comes to efficiency, data structures help the computer to run the code faster by organizing the data in a way that is easy for the computer to process.

➔A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.

The choice of a good data structure makes it possible to perform a variety of critical operations effectively.

➔ Learning data structures and algorithms enhances your overall programming abilities. You'll be able to write code that is cleaner, more efficient, and easier to maintain. Understanding how your code works at a fundamental level makes it easier to find and fix errors.

➔ Data structures are a specific way of organizing data in a specialized format on a computer so that the information can be organized, processed, stored, and retrieved quickly and effectively. They are a means of handling information, rendering the data for easy use.

➔ **Classification/Types of Data Structures:**

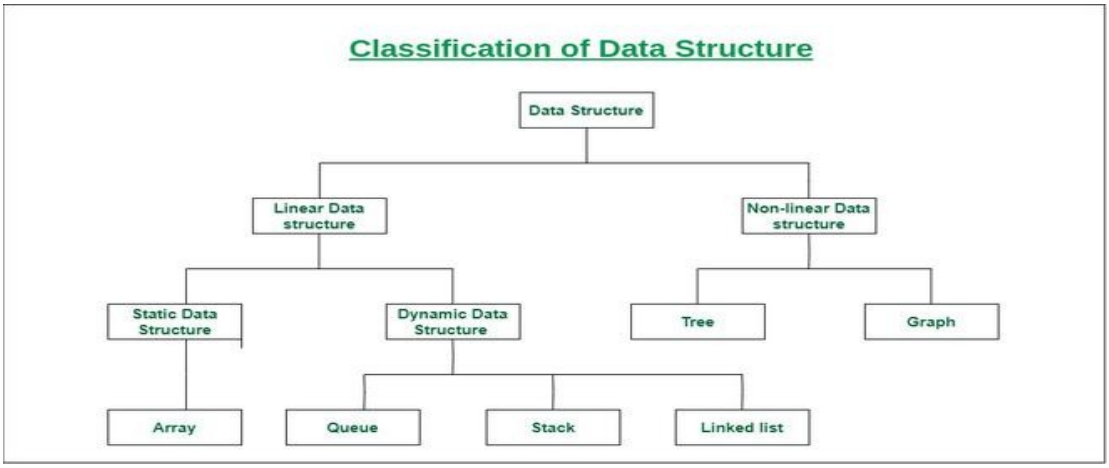
1. Linear Data Structure
2. Non-Linear Data Structure.

**Linear Data Structure:**

- Elements are arranged in one dimension, also known as linear dimension.
- Example: lists, stack, queue, etc.

**Non-Linear Data Structure**

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.




**Need Of Data Structure:**

The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.

Data structures provide an easy way of organising, retrieving, managing, and storing data.

Here is a list of the needs for data.

- Data structure modification is easy.
- It requires less time.
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database
- **Applications of Data Structures:**
- Data structures are used in various fields such as:
- Operating system
- Graphics
- Computer Design
- Blockchain
- Genetics
- Image Processing
- Simulation
-

**12 Provide a comprehensive overview of the Program Development Lifecycle?**

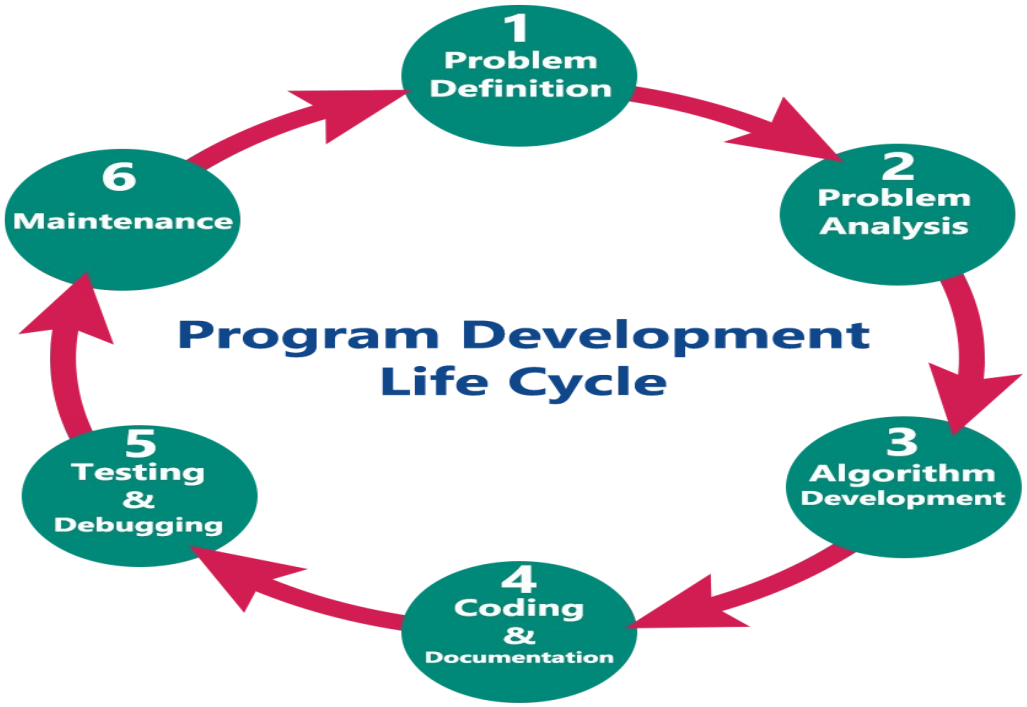
➔ The Program Development Lifecycle (PDLC) is a series of steps that are followed to develop a program in any programming language. The steps in the PDLC are:

➔ Program development is the process of creating application programs. Program development life cycle (PDLC) The process containing the five phases of program development:

**analyzing, designing, coding, debugging and testing, and implementing and maintaining application software**.

The following are six steps in the Program Development Life Cycle:

1. **Analyze the problem.** The computer user must figure out the problem, then decide how to resolve the problem - choose a program.
2. **Design the program**. A flow chart is important to use during this step of the PDLC. This is a visual diagram of the flow containing the program. This step will help you break down the problem.
3. **Code the program**. This is using the language of programming to write the lines of code. The code is called the listing or the source code. The computer user will run an object code for this step.
4. **Debug the program.** The computer user must debug. This is the process of finding the "bugs" on the computer. The bugs are important to find because this is known as errors in a program.
5. **Formalize the solution.** One must run the program to make sure there are no syntax and logic errors. Syntax are grammatical errors and logic errors are incorrect results.
6. **Document and maintain the program.** This step is the final step of gathering everything together. Internal documentation is involved in this step because it explains the reasoning one might of made a change in the program or how to write a program




➔The analysis stage includes gathering all the specific details required for a new system as well as determining the first ideas for prototypes. Developers may:

➔Define any prototype system requirements. Evaluate alternatives to existing prototypes. Perform research and analysis to determine the needs of end-users.

**Debugging:**

➔Debugging is basically making sure that a program does not have any bugs (errors) so that it can run properly without any problems.

➔Debugging is a large part of what a programmer does. The first step to debugging is done before you can actually debug the program; the program needs to be changed into machine language so that the computer can read it.

➔It is converted using a language translator. The first goal of debugging is to get rid of syntax errors and any errors that prevent the program from running. Errors that prevent the program from running are compiler errors.

**Testing/Implementation and Maintenance:**

➔Relating to getting a program up and running, many things need to happen before it can be used. One step is to test the program.

➔After the debugging process occurs, another programmer needs to test the program for any additional errors that could be involved in the background of the program.

## 13 Analyse the differences between Insertion and Bubble Sorting algorithms? Block Diagrams, providing examples programs

➔Bubble sort is a type of sorting algorithm you can use to arrange a set of values in ascending order. If you want, you can also implement bubble sort to sort the values in descending order. A real-world example of a bubble sort algorithm is how the contact list on your phone is sorted in alphabetical order.

➔Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card.

➔**Bubble sort**

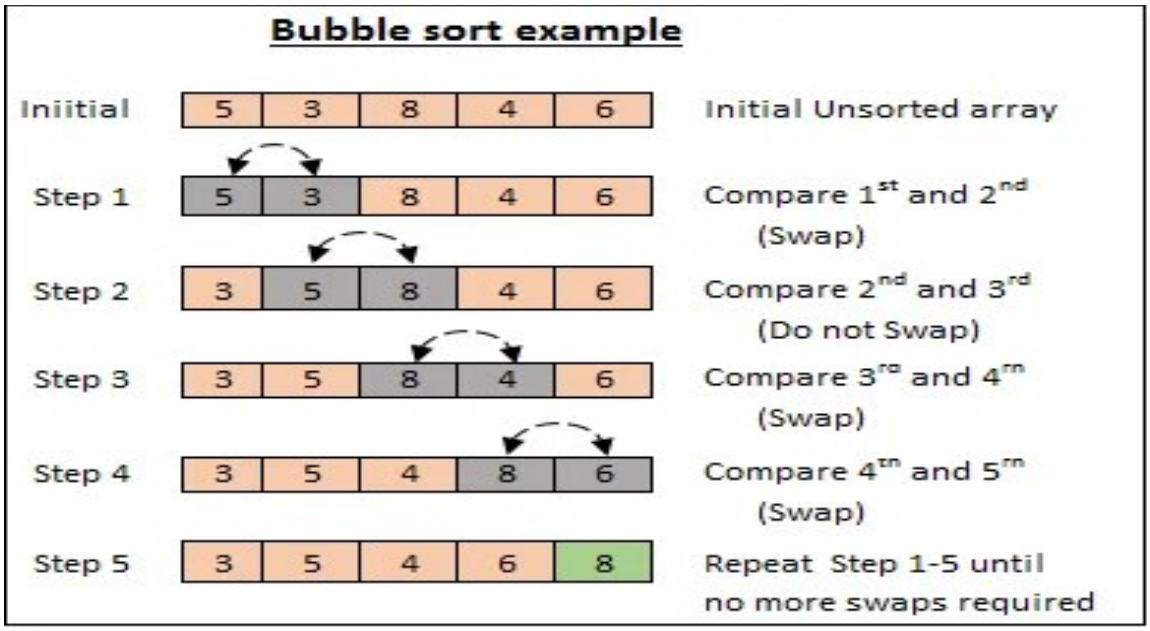
- Start at the beginning of the list.
- Compare the first value in the list with the next one up. If the first value is bigger, swap the positions of the two values.
- Move to the second value in the list. ...
- Keep going until there are no more items to compare.
- Go back to the start of the list.

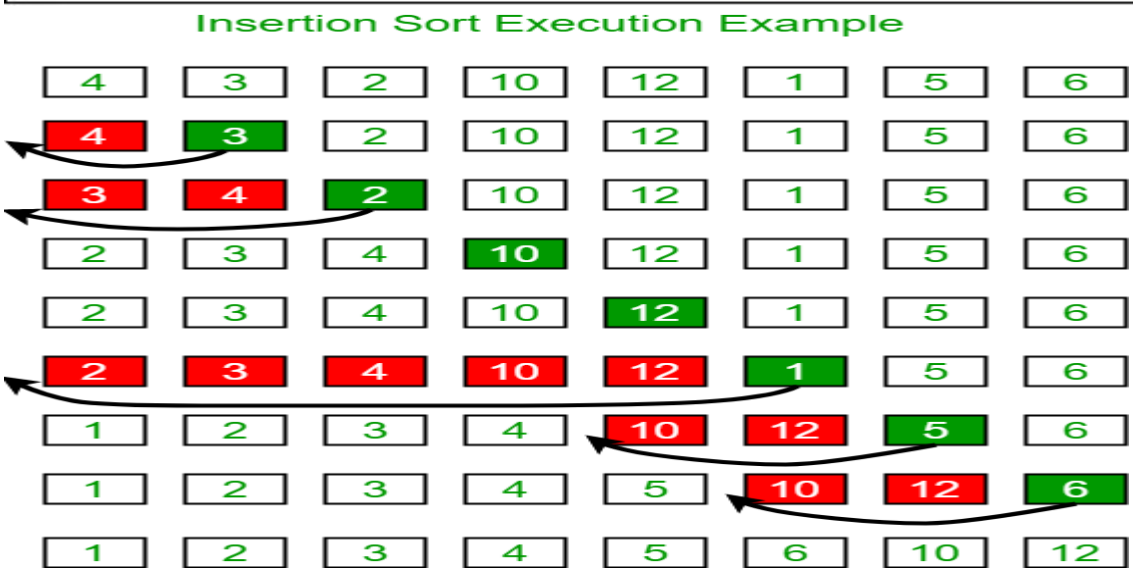
➔Insertion sort works by continually inserting each element from an unsorted subarray into a sorted subarray (hence the name “insertion” sort).

➔Which of the following real time examples is based on insertion sort? Explanation: Arranging a pack of cards mimics an insertion sort. Database scenario is an example for merge sort, arranging books is a stack and real-time systems uses quick sort.

**Insertion sort Algorithm:**

- Take any item from the unsorted part of the list.
- Compare it to each item in the already-sorted part of the list.
- If the item is found to be in-between the values of two existing sorted items, then insert it between the two items.
- Repeat the process until all the items have been sorted.




*// C program for implementation of Bubble sort*

#include <*stdio.h*>

```
void swap(int* arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Last i elements are already in place, so the loop
        // will only num n - i - 1 times
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
```

```
                swap(arr, j, j + 1);
        }
    }
}

int main() {
    int arr[] = { 6, 0, 3, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Calling bubble sort on array arr
    bubbleSort(arr, n);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

**Output**

0 3 5 6

```
// C program to implement insertion sort
#include <math.h>
#include <stdio.h>

void insertionSort(int arr[], int N) {

    // Starting from the second element
    for (int i = 1; i < N; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1], that are
          // greater than key, to one position to
          // the right of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        // Move the key to its correct position
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Unsorted array: ");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Calling insertion sort on array arr
    insertionSort(arr, N);

    printf("Sorted array: ");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

**Output**

Unsorted array: 12 11 13 5 6
Sorted array: 5 6 11 12 13

➔ **To write effective pseudocode, follow these basic principles:**

1. Use everyday language. Write in simple terms, making it easy for others to understand the algorithm.
2. Include essential details. Clearly describe the algorithm's inputs, outputs and necessary steps.
3. Keep it concise. Avoid long or complex explanations.

**Algorithm:** An algorithm is a structured method for a computer to solve a problem step by step. **Pseudocode:** is a simplified version of code written in plain language, helping to outline a program's logic before implementation. Algorithms and pseudocode are essential components of any programming language.

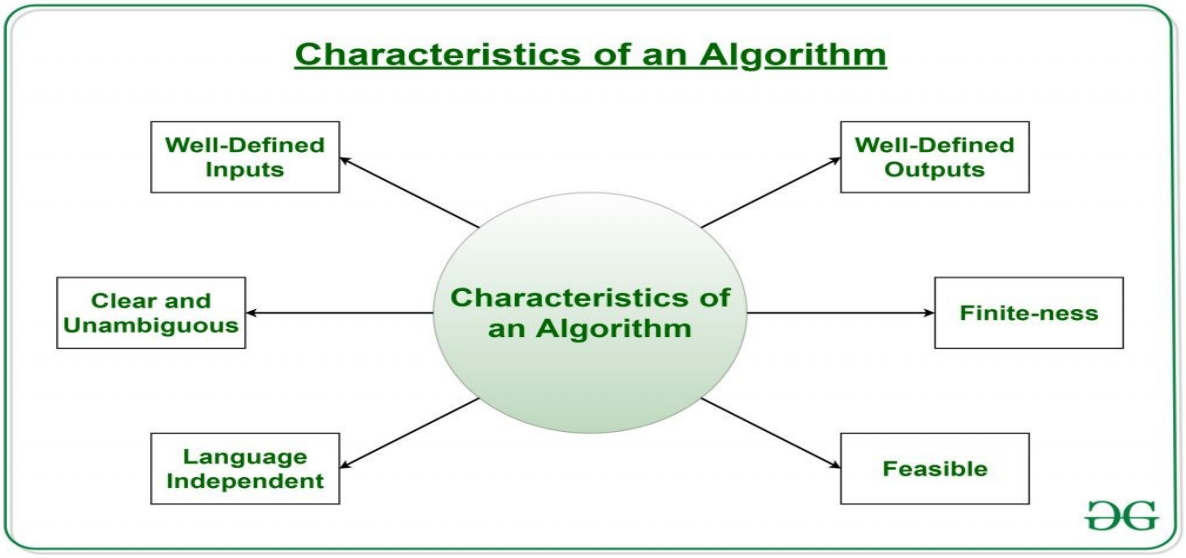
**Efficiency:** A good algorithm should perform its task quickly and use minimal resources. Correctness: It must produce the correct and accurate output for all valid inputs. **Clarity:** The algorithm should be easy to understand and comprehend, making it maintainable and modifiable.

➔Algorithms are used to find the best possible way to solve a problem, based on data storage, sorting and processing, and machine learning. In doing so, they improve the efficiency of a program. Algorithms are used in all areas of computing.

➔ A recursive function is a function that calls itself somewhere in its body. As an example, the factorial can be written as this: def factorial(n) where n is a non-negative integer:

**return 1 if n == 0**

➔ Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself. For example, we can define the operation "find your way home" as: If you are at home, stop moving. Take one step toward home.




**Find greatest integer using Recursive Algorithm:**

```
#include <stdio.h>

int max(int a, int b)
{
   return a>b?a:b ;
}

int findMaxRec(int A[], int n)
{
   if (n == 1)
      return A[0];
   return max(A[n-1], findMaxRec(A, n-1));
}
int main()
{
   int arr[] = {10, 324, 45, 90, 9808};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("Largest in given array is %d", findMaxRec(arr, n));
   return 0;
}
```

**Output**

Largest in given array is 9808

**14 Write a program to Implementation of polynomial Addition using Linked list with an output?**

➔Polynomial addition using a linked list is a process that involves adding the coefficients of like terms in two polynomials and generating a new linked list for the result:

1. **Declare variables**: Declare variables that point to the head of each linked list.
2. **Compare powers**: Compare the powers of the first polynomial in each list.

**Polynomial Addition using Linked list:**

```
#include <stdio.h>
#include <stdlib.h>

// Define a structure for a term in a polynomial
struct Term {
    int coefficient;
    int exponent;
    struct Term *next;
};

// Function prototypes
struct Term *createTerm(int coefficient, int exponent);
void insertTerm(struct Term **poly, int coefficient, int exponent);
void displayPolynomial(struct Term *poly);
struct Term *addPolynomials(struct Term *poly1, struct Term *poly2);

int main() {
    struct Term *poly1 = NULL, *poly2 = NULL, *result = NULL;
    int n1, n2, coeff, exp;

    // Input for the first polynomial
    printf("Enter the number of terms in the first polynomial: ");
    scanf("%d", &n1);

    for (int i = 0; i < n1; ++i) {
        printf("Enter coefficient and exponent for term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly1, coeff, exp);
    }

    // Input for the second polynomial
    printf("\nEnter the number of terms in the second polynomial: ");
    scanf("%d", &n2);

    for (int i = 0; i < n2; ++i) {
        printf("Enter coefficient and exponent for term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly2, coeff, exp);
    }

    // Add the polynomials
    result = addPolynomials(poly1, poly2);
```

```
    // Display the result
    printf("\nResultant Polynomial: ");
    displayPolynomial(result);

    return 0;
}

// Function to create a term in a polynomial
struct Term *createTerm(int coefficient, int exponent) {
    struct Term *newTerm = (struct Term *)malloc(sizeof(struct Term));
    newTerm->coefficient = coefficient;
    newTerm->exponent = exponent;
    newTerm->next = NULL;
    return newTerm;
}

// Function to insert a term into a polynomial
void insertTerm(struct Term **poly, int coefficient, int exponent) {
    struct Term *newTerm = createTerm(coefficient, exponent);

    if (*poly == NULL) {
        *poly = newTerm;
    } else {
        struct Term *temp = *poly;

        while (temp->next != NULL) {
            temp = temp->next;
        }

        temp->next = newTerm;
    }
}

// Function to display a polynomial
void displayPolynomial(struct Term *poly) {
    while (poly != NULL) {
        printf("%dx^%d ", poly->coefficient, poly->exponent);
        if (poly->next != NULL) {
            printf("+ ");
        }
        poly = poly->next;
    }
    printf("\n");
}

// Function to add two polynomials
```

```
struct Term *addPolynomials(struct Term *poly1, struct Term *poly2) {
   struct Term *result = NULL;

   while (poly1 != NULL || poly2 != NULL) {
      int coeff1 = (poly1 != NULL) ? poly1->coefficient : 0;
      int exp1 = (poly1 != NULL) ? poly1->exponent : 0;

      int coeff2 = (poly2 != NULL) ? poly2->coefficient : 0;
      int exp2 = (poly2 != NULL) ? poly2->exponent : 0;

      int sum_coeff = coeff1 + coeff2;
      int sum_exp = exp1; // Assuming both polynomials have the same exponents

      insertTerm(&result, sum_coeff, sum_exp);

      if (poly1 != NULL) {
         poly1 = poly1->next;
      }

      if (poly2 != NULL) {
         poly2 = poly2->next;
      }
   }

   return result;
}
```

**15 Explain the Advantages of Arrays,Types of Arrays with an Example?**

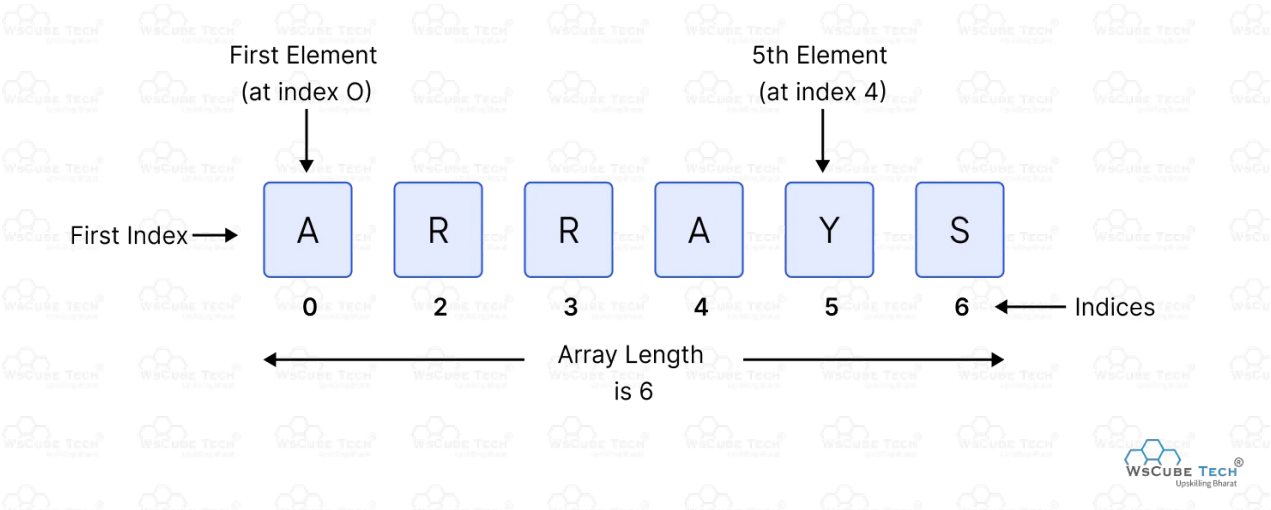
➔In an array, accessing an element is very easy by using the index number. The search process can be applied to an array easily.
➔2D Array is used to represent matrices. For any reason a user wishes to store multiple values of similar type then the Array can be used and utilized efficiently.
➔An array is a group of similar elements or data items of the same type collected at contiguous memory locations.
➔ In simple words, we can say that in computer programming, arrays are generally used to organize the same type of data.
➔arrays are faster than lists, because arrays are created with a specified length. Because of this, each element of the array are next to each other in memory.
➔The advantage of ArrayList is faster random access due to direct indexing, making it more suitable for scenarios where quick element retrieval is critical. In contrast, LinkedList is advantageous for efficient insertions and deletions at any position within the list.
➔There are two common types of arrays: linear arrays and associative arrays. Linear arrays store elements in a linear order, meaning their index begins from 0 and goes up incrementally as new elements are added. Associative arrays store elements that keys can access and their index is not based on linear order

**Types of Arrays in Data Structure**

Below are the different types of arrays in data structures:

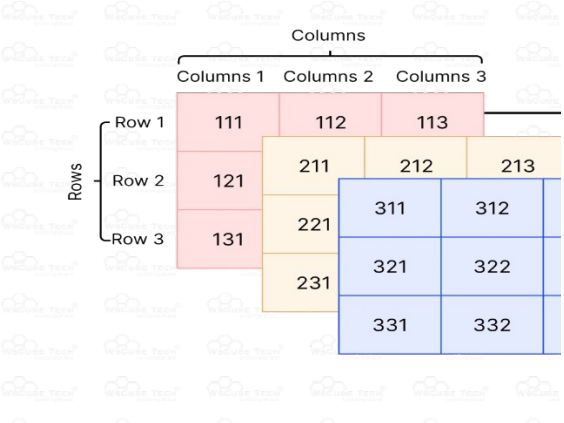
**1. One-Dimensional Arrays (1D Arrays)**

These are the simplest form of arrays, consisting of a single line of elements. Each element in a 1D array is accessed by a single index representing its position within the array.




**2.Multi-Dimensional Arrays**

These are the arrays with more than two dimensions. The multi-dimensional arrays in data structures can be used in scenarios requiring complex data representation.




**Advantages of Arrays in DSA**

**Following are the benefits of using arrays in data structures and algorithms:**

- **Random Access:** Arrays provide constant time access to any element, as each element can be accessed directly using its index.
- **Efficiency:** Storing elements contiguously in memory makes arrays very memory-efficient and optimizes performance.

- **Simplicity:** Arrays have a simple and straightforward structure, making them easy to understand and implement.
- **Basis for Other Structures: Arrays** serve as the foundational building block for constructing other complex data structures like **heaps**, **hash tables**, **stacks**, and **queues**.
- **Good for Fixed-Size Collections:** When the size of the dataset is known and not expected to change, arrays are an excellent choice due to their static nature.
- **Performance in Iterative Processes:** Due to their contiguous memory allocation, arrays allow fast iteration over elements, making them ideal for loops and algorithmic processing.
- **Cache-Friendliness:** The contiguous allocation of memory makes arrays cache-friendly, enhancing performance by reducing cache misses during operations that access array elements sequentially.
- **Ease of Manipulation:** Basic operations like sorting and searching are well-understood and easily implemented on arrays, benefiting from direct element access.
- **Predictable Memory Allocation:** Memory for arrays is allocated at the time of array creation, which simplifies memory management in environments with limited resources or where performance predictability is critical.

  **Example:**
  Insert the value 25 at the third position in an array.
  elements = [10, 20, 30, 40, 50]
  elements.insert(2, 25)  # Insert 25 at index 2
  print(elements)  # Outputs [10, 20, 25, 30, 40, 50]

**16 Describe the Linear Search Algorithm,Write an Example program of Linear Search?**

➔In linear search, the list is simply traversed, and each element in the list is matched with the element whose location needs to be found.
When the searching process is done, it returns the location of the element, else the algorithm returns NULL. It has high complexity if the linear search is O(n).
➔Linear search is used on a collections of items. It relies on the technique of traversing a list from start to end by exploring properties of all the elements that are found on the way.
➔ For example, consider an array of integers of size . You should find and print the position of all the elements with value.
➔**Algorithm for Linear Search Algorithm:**

The algorithm for linear search can be broken down into the following steps:

- **Start:** Begin at the first element of the collection of elements.
- **Compare:** Compare the current element with the desired element.
- **Found:** If the current element is equal to the desired element, return true or index to the current element.
- **Move:** Otherwise, move to the next element in the collection.
- **Repeat:** Repeat steps 2-4 until we have reached the end of collection.
- **Not found:** If the end of the collection is reached without finding the desired element, return that the desired element is not in the array.

```
// C program to implement linear search using loop
#include <stdio.h>

```

```c
int linearSearch(int* arr, int n, int key) {

    // Starting the loop and looking for the key in arr
    for (int i = 0; i < n; i++) {

        // If key is found, return key
        if (arr[i] == key) {
            return i;
        }
    }

    // If key is not found, return some value to indicate
    // end
    return -1;
}

int main() {
    int arr[] = { 10, 50, 30, 70, 80, 60, 20, 90, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    // Calling linearSearch() for arr with key = 43
    int i = linearSearch(arr, n, key);

    // printing result based on value returned by
    // linearSearch()
    if (i == -1)
        printf("Key Not Found");
    else
        printf("Key Found at Index: %d", i);

    return 0;
}
```

**Output**

```
Key Found at Index: 2
```

**17 Discuss the Binary Search Algorithm ,Write an Example Program of Binary search?**

Binary search is a search algorithm that finds a target value in a sorted array by repeatedly dividing the array in half:

➔ **How it works:**

Binary search compares the target value to the middle element of the array. If the values are not equal, the algorithm eliminates the half of the array that cannot contain the target value.

➔Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

➔ **Efficient:** Binary search has a time complexity of O(log n), which makes it very efficient for searching large sorted arrays. Simple to implement:

Binary search is relatively easy to implement and understand. Versatile: Binary search can be used in a wide variety of applications.

```
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
if (high >= low) {
int mid = low + (high - low) / 2;
// If found at mid, then return it
if (array[mid] == x)
return mid;
// Search the left half
if (array[mid] > x)
return binarySearch(array, x, low, mid - 1);
// Search the right half
return binarySearch(array, x, mid + 1, high);
}
return -1;
}
int main(void) {
int array[] = {3, 4, 5, 6, 7, 8, 9};
int n = sizeof(array) / sizeof(array[0]);
int x = 4;
int result = binarySearch(array, x, 0, n - 1);
if (result == -1)
printf("Not found");
else
printf("Element is found at index %d", result);
}
```

**18 Explain the Merge Sort, Write a program to Implementations of merge sort.?**

Merge sort is one of the most efficient sorting algorithms. It is based on the divide-and-conquer strategy. Merge sort continuously cuts down a list into multiple sublists until each has only one item, then merges those sublists into a sorted list.

➔ The merge sort in C is a stable sorting algorithm that follows the divide and conquer technique to sort an array in ascending order.

➔ The merge sort program in C divides an array into two halves and then merges the two halves in sorted order.

```
#include <stdio.h>

// Function to merge two halves
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1; // Size of left subarray
    int n2 = right - mid;    // Size of right subarray

    int L[n1], R[n2];
```

```
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0; j = 0; k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function that implements MergeSort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
```

```
    printf("\n");
}

// Main function
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arrSize = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, arrSize);

    mergeSort(arr, 0, arrSize - 1);

    printf("Sorted array: \n");
    printArray(arr, arrSize);
    return 0;
}
```

**19 Explain the Quick Sort, Write a program to Implementations of Quick sort.?**

## Quick sort:

Quicksort is a highly efficient sorting technique that divides a large data array into smaller ones. A vast array is divided into two arrays, one containing values smaller than the provided value, say pivot, on which the partition is based.

The other contains values greater than the pivot value.

Quick Sort is usually the fastest sorting algorithm. Its performance is measured most of the time in O(N × log N).

Quicksort has many advantages, such as being fast, easy to implement, and in-place, which means that it does not require extra space.

Quicksort also has some disadvantages, such as being unstable, sensitive to the choice of the pivot, and vulnerable to the worst case.

```
// Quick sort in C

#include <stdio.h>

// function to swap elements
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {
```

```
  int pivot = array[high];
  int i = (low - 1);

  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      i++;
      swap(&array[i], &array[j]);
    }
  }
  swap(&array[i + 1], &array[high]);

  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {

    int pi = partition(array, low, high);
    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
  }
}

// function to print array elements
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// main function
int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};

  int n = sizeof(data) / sizeof(data[0]);

  printf("Unsorted Array\n");
  printArray(data, n);
  quickSort(data, 0, n - 1);

  printf("Sorted array in ascending order: \n");
  printArray(data, n);
}
```

**20. Discuss can linear and non-linear data structures be differentiated based on the storage of the elements in memory, Justify your answer.**

Linear data structures have a single level of elements arranged sequentially, while non-linear data structures have multiple levels arranged hierarchically or networked. Linear data structures use contiguous or non-contiguous memory, whereas non-linear structures are typically non-contiguous.

➔ Linear structures arrange data in a linear sequence, such as found in an array, list, or queue. In nonlinear structures, the data doesn't form a sequence but instead connects to two or more information items, like in a tree or graph.

| S.NO | Linear Data Structure | Non-linear Data Structure |
|---|---|---|
| 1. | In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent. | In a non-linear data structure, data elements are attached in hierarchically manner. |
| 2. | In linear data structure, single level is involved. | Whereas in non-linear data structure, multiple levels are involved. |
| 3. | Its implementation is easy in comparison to non-linear data structure. | While its implementation is complex in comparison to linear data structure. |
| 4. | In linear data structure, data elements can be traversed in a single run only. | While in non-linear data structure, data elements can’t be traversed in a single run only. |
| 5. | In a linear data structure, memory is not utilized in an efficient way. | While in a non-linear data structure, memory is utilized in an efficient way. |
| 6. | Its examples are: array, stack, queue, linked list, etc. | While its examples are: trees and graphs. |
| 7. | Applications of linear data structures are mainly in application software development. | Applications of non-linear data structures are in Artificial Intelligence and image processing. |
| 8. | Linear data structures are useful for simple data storage and manipulation. | Non-linear data structures are useful for representing complex relationships and data hierarchies, such as in social networks, file systems, or computer |

| S.NO | Linear Data Structure | Non-linear Data Structure |
|---|---|---|
| | | networks. |
| 9. | Performance is usually good for simple operations like adding or removing at the ends, but slower for operations like searching or removing elements in the middle. | Performance can vary depending on the structure and the operation, but can be optimized for specific operations. |

**1. Array**

The array is a type of data structure that stores elements of the same type. These are the most basic and fundamental data structures. Data stored in each position of an array is given a positive value called the index of the element. The index helps in identifying the location of the elements in an array.

If supposedly we have to store some data i.e. the price of ten cars, then we can create a structure of an array and store all the integers together. This doesn't need creating ten separate integer variables. Therefore, the lines in a code are reduced and memory is saved. The index value starts with 0 for the first element in the case of an array.

**2. Stack**

The data structure follows the rule of LIFO (Last In-First Out) where the data last added element is removed first. Push operation is used for adding an element of data on a stack and the pop operation is used for deleting the data from the stack. This can be explained by the example of books stacked together. In order to access the last book, all the books placed on top of the last book have to be safely removed.

**3. Queue**

This structure is almost similar to the stack as the data is stored sequentially. The difference is that the queue data structure follows FIFO which is the rule of First In-First Out where the first added element is to exit the queue first. Front and rear are the two terms to be used in a queue.

Enqueue is the insertion operation and dequeue is the deletion operation. The former is performed at the end of the queue and the latter is performed at the start end. The data structure might be explained with the example of people queuing up to ride a bus. The first person in the line will get the chance to exit the queue while the last person will be the last to exit.

**4. Linked List**

Linked lists are the types where the data is stored in the form of nodes which consist of an element of data and a pointer. The use of the pointer is that it points or directs to the node which is next to the element in the sequence. The data stored in a linked list might be of any form, strings, numbers, or characters. Both sorted and unsorted data can be stored in a linked list along with unique or duplicate elements.

**5. Hash Tables**

These types can be implemented as linear or non-linear data structures. The data structures consist of key-value pairs.

**Non-linear Structure:**

Data structures where data elements are not arranged sequentially or linearly are called **non-**

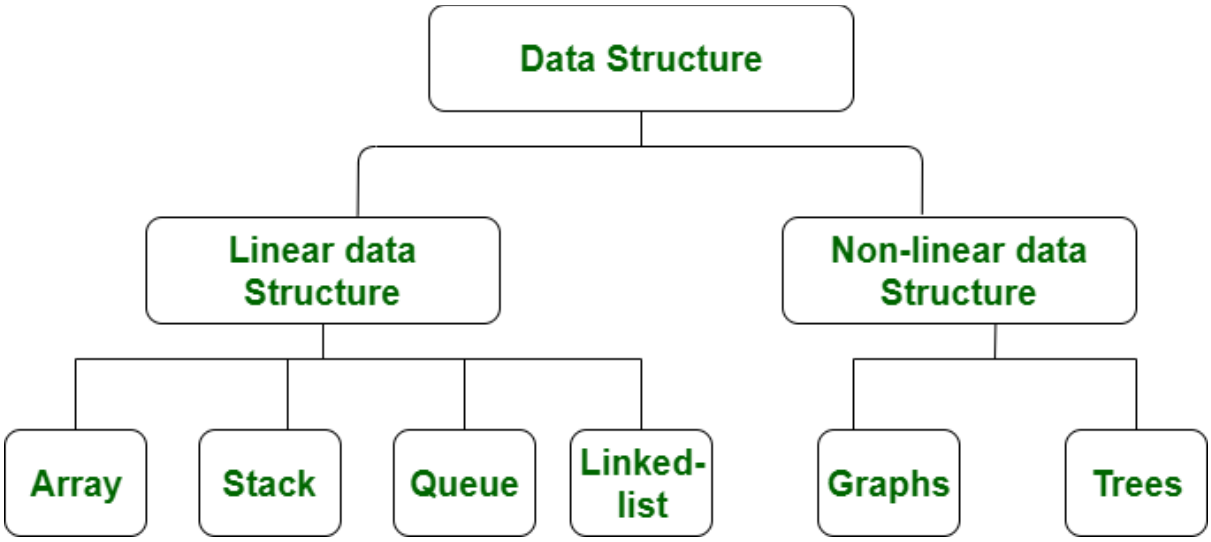
**linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

**1. Trees**

A tree data structure consists of various nodes linked together. The structure of a tree is hierarchical that forms a relationship like that of the parent and a child. The structure of the tree is formed in a way that there is one connection for every parent-child node relationship. Only one path should exist between the root to a node in the tree. Various types of trees are present based on their structures like AVL tree, binary tree, binary search tree, etc.

**2. Graph**

Graphs are those types of non-linear data structures which consist of a definite quantity of vertices and edges. The vertices or the nodes are involved in storing data and the edges show the vertices relationship. The difference between a graph to a tree is that in a graph there are no specific rules for the connection of nodes. Real-life problems like social networks, telephone networks, etc. can be represented through the graphs.




Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**.

In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only.

Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.

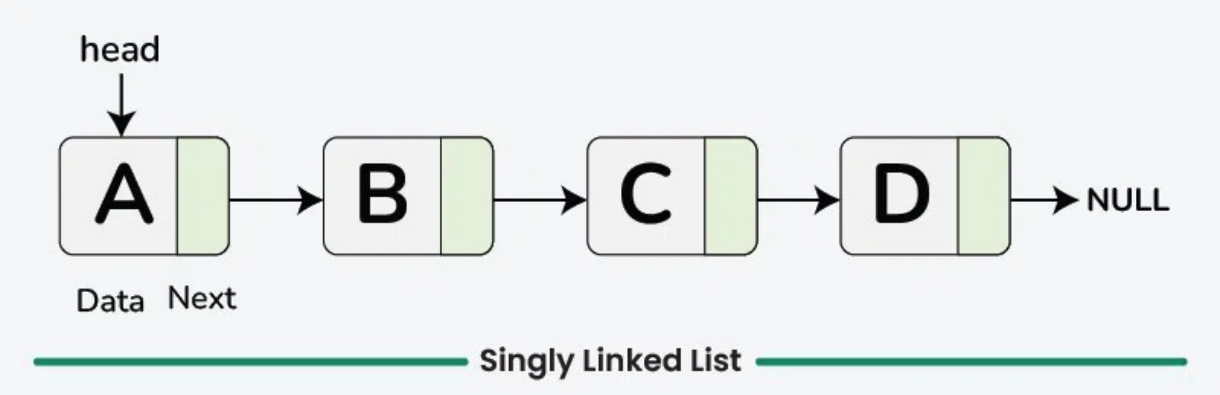
Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

**21. Explain the Linked list, Operation on a Different types of Linked list representation using Block Diagrams.**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

➔ **1. Singly Linked List**

*Singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.*
The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way. Below is the image for the same:




```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    // Data part of the node
    int data;

    // Pointer to the next node in the list
    struct Node* next;
};
// C program to illustrate creation
// and traversal of Singly Linked List

#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};
```

```
struct Node* createNode(int data) {
   struct Node* node =
     (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->next = NULL;
   return node;
}

void printList(struct Node* n) {

   // Iterate till n reaches NULL
   while (n != NULL) {
      printf("%d ", n->data);
      n = n->next;
   }
}

int main() {

   struct Node* head = NULL;
   struct Node* second = NULL;
   struct Node* third = NULL;

   //Linked List 1 -> 2 -> 3
   head = createNode(1);
   second = createNode(2);
   third = createNode(3);

   head->next = second;

   second->next = third;

   printList(head);

   return 0;
}
```

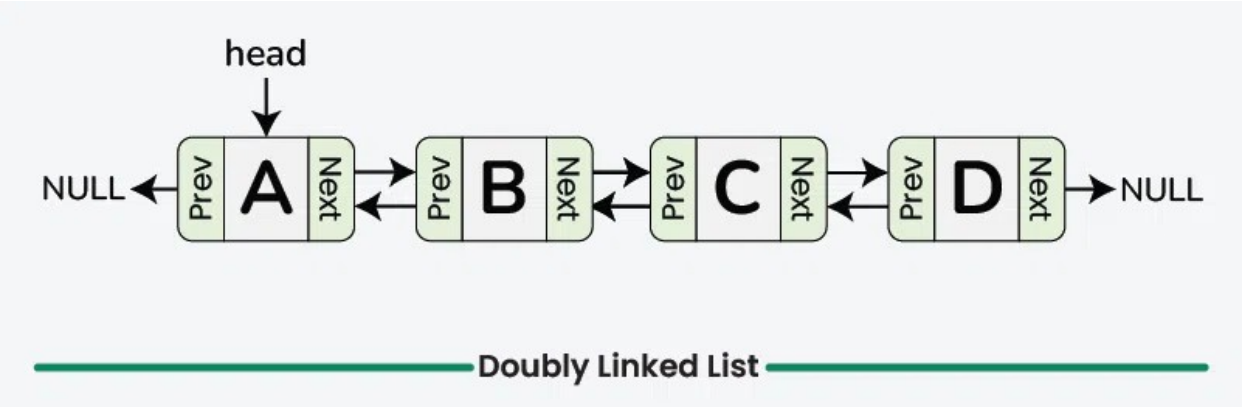
**Output**

1 2 3

**2. Doubly Linked List**

*A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the **next** as well as the **previous** node in sequence.*

Therefore, it contains three parts of data, a pointer to the **next** node, and a pointer to the **previous** node. This would enable us to traverse the list in the backward direction as well.




```
// Define the Node structure
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

struct Node* createNode(int data) {
    struct Node* newNode =
      (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}


// C  program to illustrate creation
// and traversal of doubly Linked List
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};


void forwardTraversal(struct Node* head) {
    struct Node* curr = head;
    while (curr != NULL) {
        printf("%d ", curr->data);
        curr = curr->next;
    }
    printf("\n");
```

```
}

void backwardTraversal(struct Node* tail) {
   struct Node* curr = tail;
   while (curr != NULL) {
      printf("%d ", curr->data);
      curr = curr->prev;
   }
   printf("\n");
}

struct Node* createNode(int data) {
   struct Node* newNode =
     (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   newNode->prev = NULL;
   return newNode;
}


int main() {

     //Linked List 1 <-> 2 <-> 3
   struct Node* head = createNode(1);
   struct Node* second = createNode(2);
   struct Node* third = createNode(3);

   head->next = second;
   second->prev = head;
   second->next = third;
   third->prev = second;

   printf("Forward Traversal:\n");
   forwardTraversal(head);

   printf("Backward Traversal:\n");
   backwardTraversal(third);

   return 0;
}
```

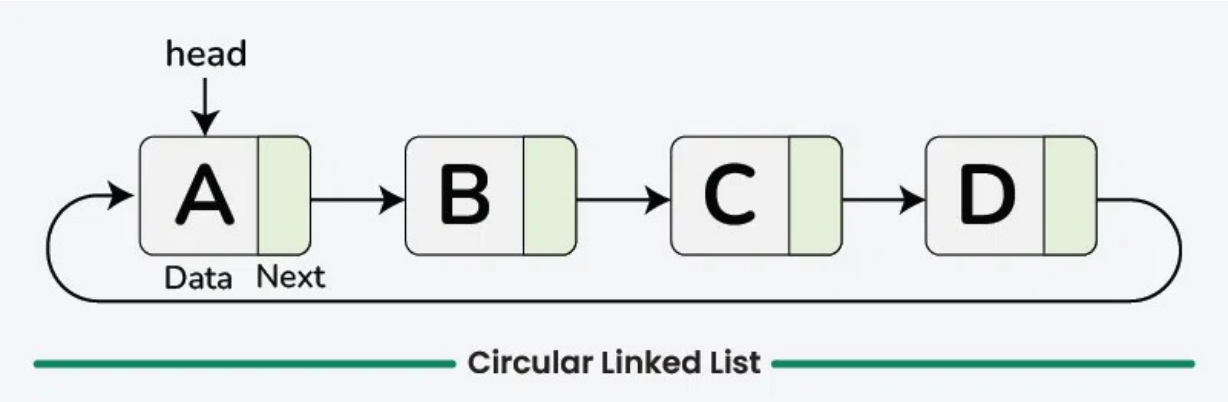
**Output**

Forward Traversal:
1 2 3
Backward Traversal:
3 2 1

### 3. Circular Linked List

*A circular linked list is a type of linked list in which the last node's next pointer points back to the first node of the list, creating a circular structure. This design allows for continuous traversal of the list, as there is no null to end the list.*

While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end. Below is the image for the same:




```
// Define the Node structure
struct Node {
    int data;
    struct Node *next;
};

struct Node *createNode(int value) {
    struct Node *newNode =
      (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}


/ C  program to illustrate creation
// and traversal of Circular Linked List
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};

void printList(struct Node *last) {
```

```
    if (last == NULL) return;

    struct Node *head = last->next;
    while (1){
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next)
            break;
    }
    printf("\n");
}

struct Node *createNode(int value) {
    struct Node *newNode =
      (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}


int main() {

    // Create circular linked list: 2, 3, 4
    struct Node *first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);
    struct Node *last = first->next->next;
    last->next = first;
    printList(last);

    return 0;
}
```

**Output**

2 3 4

**4. Doubly Circular linked list**

Doubly Circular linked list or a circular **two-way** linked list is a complex type of linked list that contains a pointer to the **next** as well as the **previous** node in the sequence. The difference between th**e doubly linked** and **circular doubly list** is the same as that between a **singly linked list** and a **circular linked list**. The circular doubly linked list does not contain **null** in the **previous** field of the first node.

**5.Multilevel Linked List :**

A **multilevel** linked list is an advanced data structure designed to represent **hierarchical** relationships among elements.

Each node in this structure contains **three** main components: **data**, a **next** pointer, and a **child** pointer. The **next** pointer links to the **next** node in the **same level** of the list, allowing for **linear** traversal within that level. The **child** pointer, on the other hand, points to a **sub-list o**r nested linked list, creating a hierarchical or **tree-like** structure. This enables each node to have its own sub-list of nodes, allowing for complex nested relationships. Please refer to Multi Linked List for Implementation.

**6. Skip-List :**

A **skip list** is a data structure that allows for efficient **search**, **insertion** and **deletion** of elements in a **sorted** list. It is a probabilistic data structure, meaning that its average **time complexity** is determined through a **probabilistic** analysis. Skip lists are implemented using a technique called "**coin flipping."** In this technique, a **random** number is generated for each insertion to determine the **number of layer**s the new element will occupy. This means that, on average, each element will be in **log(n)** layers, where **n** is the number of elements in the bottom layer.