303105201 - Design of Data Structures

UNIT-1

Introduction:

->Data Structures

Classifications (Primitive & Non-Primitive)

Data structure Operations

Review of Arrays

Structures

Self-Referential Structures and Unions

->Pointers and Dynamic Memory Allocation Functions

->Representation of Linear Arrays in Memory

Dynamically allocated arrays

->Performance analysis of an algorithm and space and time complexities

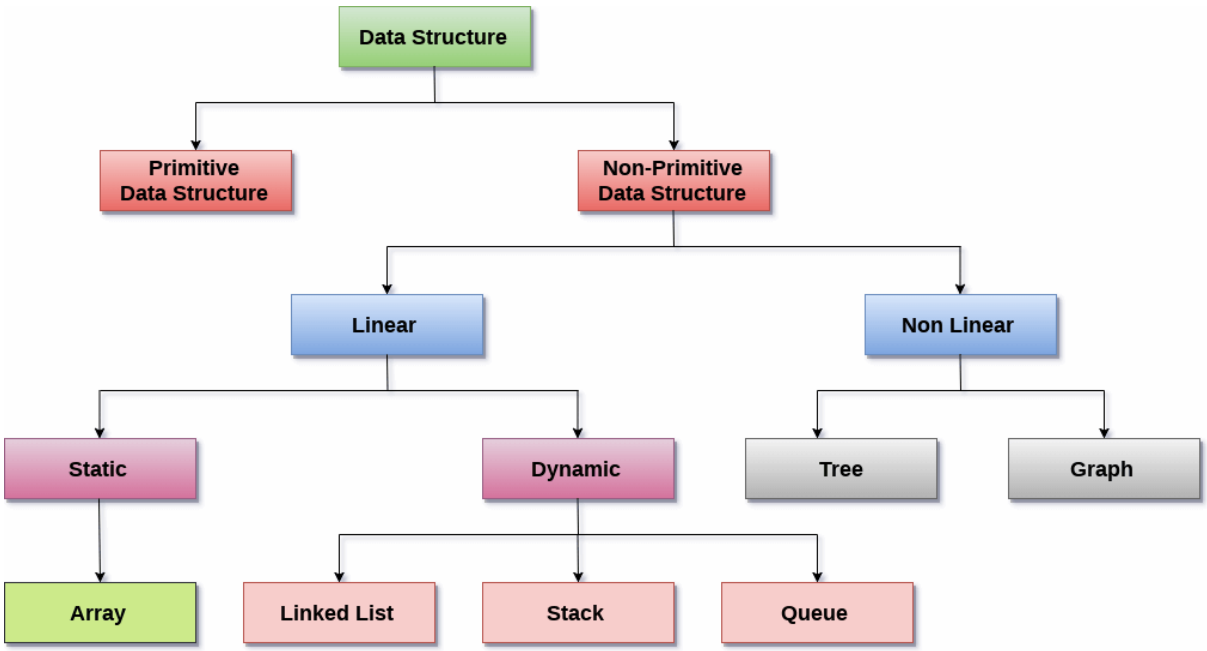
Data Structures:

Data structures can be implemented using various techniques such as arrays, structures, pointers, and dynamic memory allocation. Examples include arrays, linked lists, stacks, queues, trees, graphs, and hash tables.

Classifications (Primitive & Non-Primitive):

Primitive Data Types: These are basic data types provided by C and include int, char, float, double, void, etc.

Non-Primitive Data Types: These are user-defined data types created using primitive data types and include arrays, structures, pointers, and functions.



Data Structure Operations:

In C, data structure operations are typically implemented through functions that manipulate the underlying data structure. Common operations include:

Insertion: Adding elements to the structure.

Deletion: Removing elements from the structure.

Traversal: Iterating through the elements of the structure.

Searching: Finding specific elements within the structure.

Sorting: Arranging elements in a particular order.

Accessing: Retrieving elements from the structure.

Review of Arrays:

Arrays in C are collections of elements of the same data type stored in contiguous memory locations. They are declared using square brackets [] and accessed using zero-based indexing.

For *Example:*

```
int numbers[5]; // Declaration of an integer array with 5 elements

numbers[0] = 10; // Assigning a value to the first element
```

Structures:

Structures allow grouping together variables of different data types under a single name. They are defined using the struct keyword.

For *Example:*

```
struct Person {
    char name[50];
    int age;
    float salary;
};
```

You can then declare variables of this structure type and access their members using the dot (.) operator.

Self-Referential Structures and Unions:

Self-Referential Structures:

Self-referential structures are often used to implement linked data structures such as linked lists and trees.

For *Example*:

```
struct Node {
   int data;
   struct Node* next; // Self-referential pointer to another Node
};
```

Unions:

Unions allow different data types to be stored in the same memory location. However, only one member of the union can hold a value at a time.

For *Example:*

```
union Data {
   int i;
   float f;
   char str[20];
};
```

Pointers and Dynamic Memory Allocation Functions:

Pointers:

Pointers are variables that store memory addresses. They enable dynamic memory allocation, passing by reference, and working with complex data structures. In C, pointers are declared using the * operator.

Dynamic Memory Allocation Functions:

In C, dynamic memory allocation is achieved using malloc, calloc, realloc, and free functions.

*/*Note:* Here, Ptr stands for pointer variable*/

malloc(size_t size):

Allocates a block of memory of the specified size in bytes.

Syntax: Ptr=(data_tye*)malloc(n*sizeof(data_type));

calloc(size_t num, size_t size):

Allocates an array of elements, each of the specified size, and initializes them to zero.

Syntax: Ptr=(data_tye*)calloc(n,sizeof(data_type));

realloc(void ptr, size_t size):*

Resizes the previously allocated memory block pointed to by ptr.

Syntax: Ptr=(data_tye*)malloc(n*sizeof(data_type)); or

Ptr=(data_tye*)calloc(n,sizeof(data_type));

Ptr=(data_tye*)realloc(Ptr,n,sizeof(data_type));

free(void ptr):*

Deallocates the memory block previously allocated by malloc, calloc, or realloc.

Syntax: free(ptr);

Representation of Linear Arrays in Memory:

Arrays are of three types: 1D array (1dimentional array)

2D array (2dimentional array)

3D array (3dimentional array)

Static Arrays:

Static arrays are allocated memory at compile-time and are stored in contiguous memory locations. The address of the first element serves as the base address.

Dynamically Allocated Arrays:

These arrays are created using dynamic memory allocation functions (malloc, calloc, etc.). They are stored in the heap memory and are accessed through pointers. The size of dynamically allocated arrays can be determined at runtime.

Performance Analysis of an Algorithm and Space and Time Complexities:

Performance Analysis: Performance analysis involves evaluating the efficiency of an algorithm in terms of time and space.

Time Complexity: Time complexity measures the amount of time an algorithm takes to run as a function of the size of its input. It is often expressed using Big O notation (e.g., O(n), O(n^2)).

Space Complexity: Space complexity measures the amount of memory an algorithm uses as a function of the size of its input. Like time complexity, it is also expressed using Big O notation.

In C, you can analyse the performance of an algorithm by implementing it and measuring its execution time using profiling tools or by analysing its code complexity theoretically.

Example:

An example of dynamically allocating an array of integers, filling it with values, and then freeing the allocated memory:

```
#include <stdio.h>

#include <stdlib.h>

int main() {
   int n,i;
   printf("Enter the size of the array: ");
   scanf("%d", &n);
   // Dynamically allocate memory for the array
   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed.");
      return 1;
   }
// Fill the array with values
   for (i = 0; i < n; i++) {
      arr[i] = i * 10;
   }
   // Print the array elements
   printf("Array elements:\n");
   for (i = 0; i < n; i++) {
```

```
        printf("%d ", arr[i]);
    }
    printf("\n");
    // Free the allocated memory
    free(arr);
    return 0;
}
```

Output: