UNIT 1
Introduction

Prof. Keval Mehta,
Sr. Cyber Security Trainer
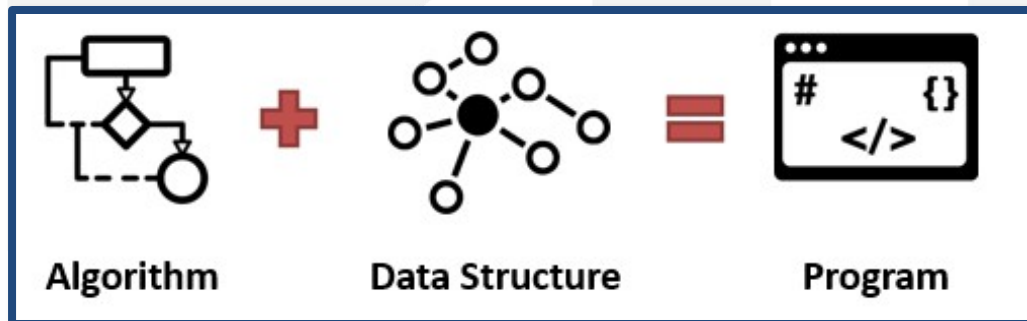Computer Science & Engineering

Parul® University

Topic-1

Introduction

Introduction

- "Data Structures and Algorithms" is one of the classic, core topics of Computer Science.

- Data structures and algorithms are central to the development of good quality computer programs.

- **Program= algorithm + Data Structure**

Algorithm + Data Structure = Program

What is Algorithms?

- In mathematics and computer science, an **algorithm** is a set of instructions, typically to solve a class of problems or perform a computation.
- Algorithm is a step by step procedure to solve a particular function.

Introduction

- It is important for every Computer Science student to understand the concept of **Information** and *how it is organized or how it can be utilized*.

- **What is Information?**

    If we arrange some data in an appropriate sequence, then it forms a Structure and gives us a meaning. This meaning is called *Information*

- Two things in Information: One is **Data** and the other is **Structure** .

Introduction

What is Data?

Facts and statistics collected together for reference or analysis.

What is Data Structure?

A data structure is a systematic way of organizing and accessing data.

- A *data structure* tries to structure data!
    - Usually more than one piece of data
    - Should define legal operations on the data
    - The data might be grouped together

Introduction

- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.

- It plays an important role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

- That means, **algorithm** is a set of instruction written to carry out certain tasks & the **data structure** is the way of organizing the data with their logical relationship retained.

- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

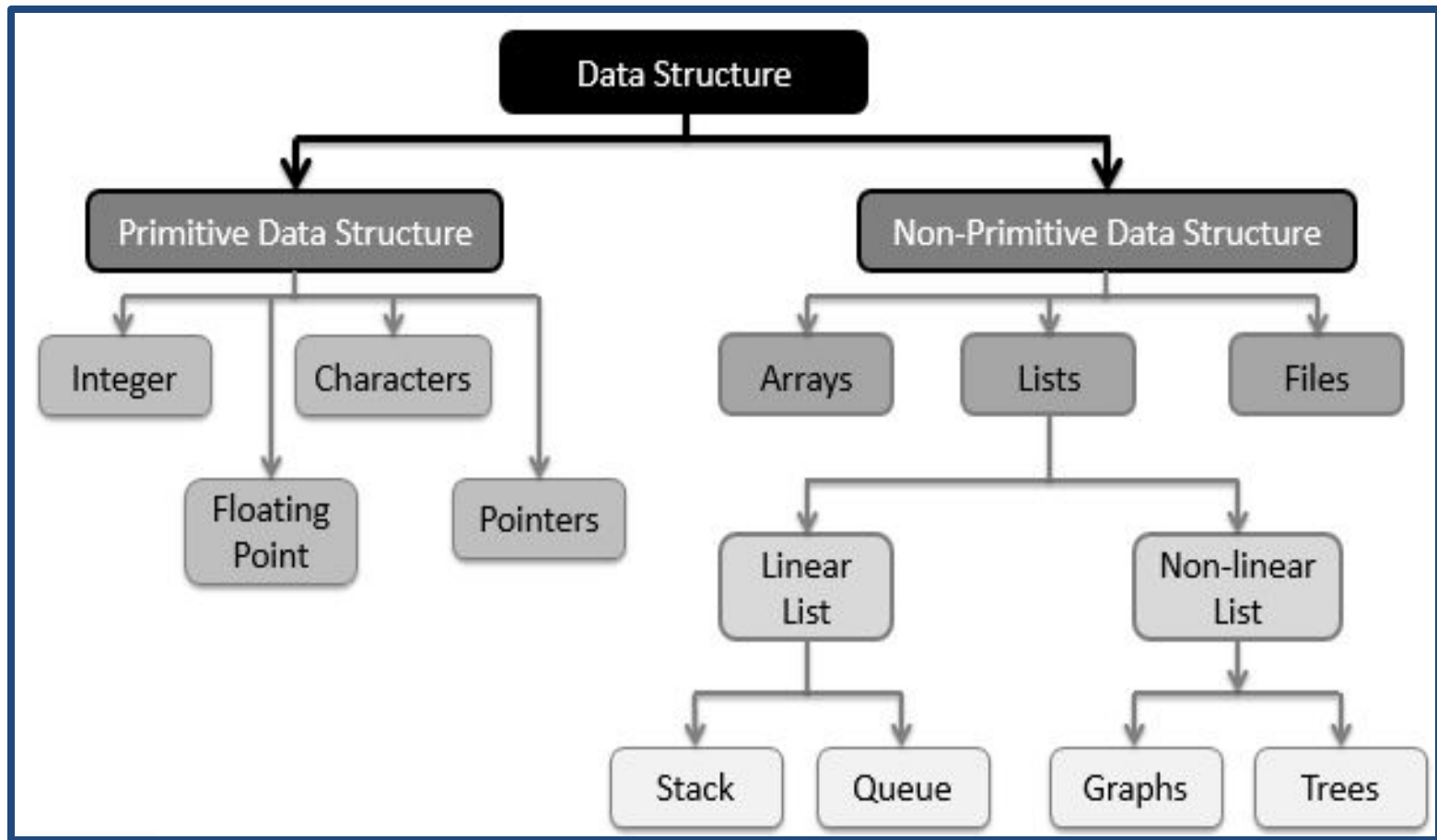- *Therefore algorithm and its associated data structures form a program.*

Topic-2

Classification of Data Structure

Classification of Data Structure

Image source : Google

Classification of Data Structure

Data Structures are normally classified into two broad categories

1.   Primitive Data Structure

2.   Non-primitive data Structure
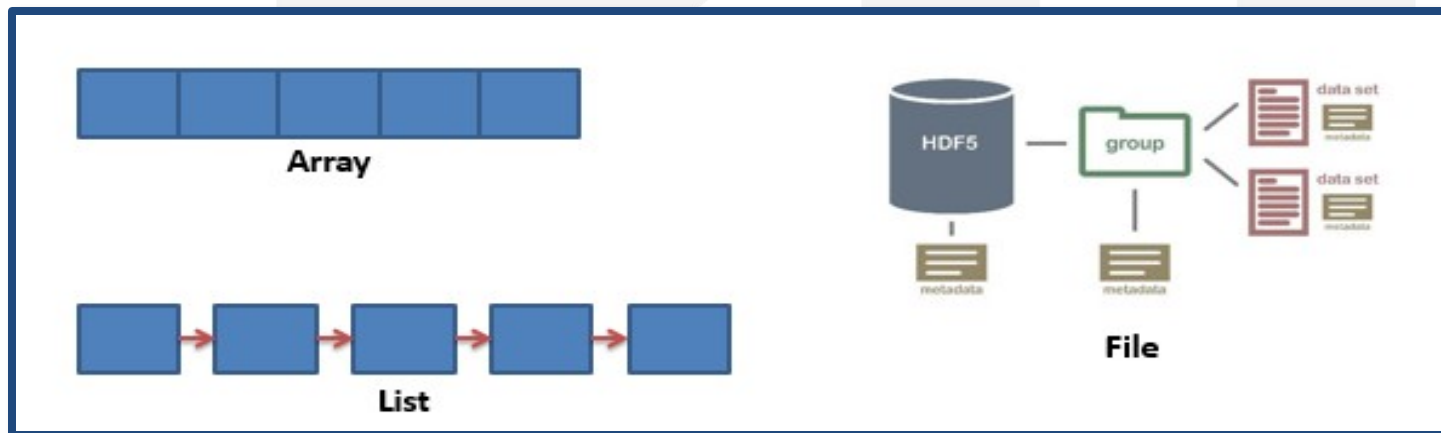
Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.

  - It have different representations on different computers.

  - **Integer**: allows all values without fraction part.

  - **Float**: used for storing fractional numbers.

  - **Character**: used for character values.

  - **Pointer**: holds memory address of another variable

Non Primitive Data Structure

- These are derived from primitive data structures.

- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.

- Examples of Non-primitive data type are Array, List, and File etc.

- A Non-primitive data type is further divided into Linear and Non-Linear data structure
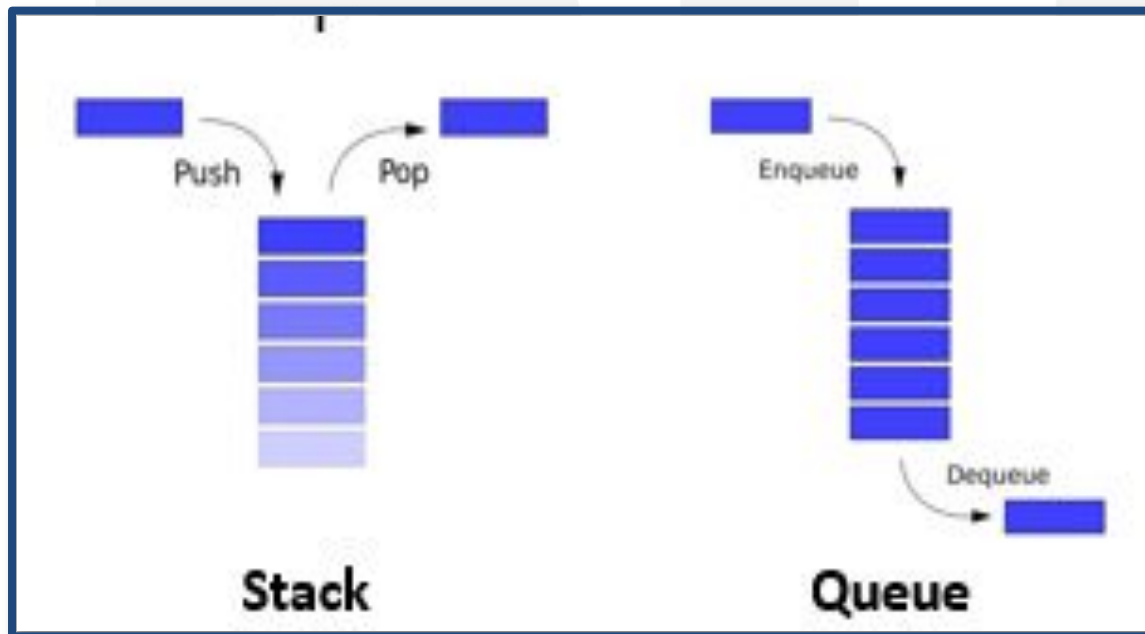
Non Primitive Data Structure

- **Array:** An array is a fixed-size sequenced collection of elements of the same data type.

- **List:** An ordered set containing variable number of elements is called as Lists.

- **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Image source : Google

Non Primitive Data Structure : Linear data structures

- Data is arranged in such a way that after one element we have just one more element that is, a single element is connected to just one more element after it.

- Examples of Linear Data Structure are Stack and Queue, linked list.

Image source : Google

Non Primitive Data Structure : Linear data structures

•**Stack**: Stack is ordered linear data structure which is modified form of an array. a data structure in which insertion and deletion operations are performed at one end only. Stack is also called as Last in First out (LIFO) data structure.

Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue. Queue is also called as First in First out (FIFO) data structure.

Non Primitive Data Structure : Linear data structures

• If after one element, we have connection to multiple elements then such data structures are called as non linear data structures.

• Examples of Non-linear Data Structure are Tree and Graph.

Tree   Graph

Image source : Google
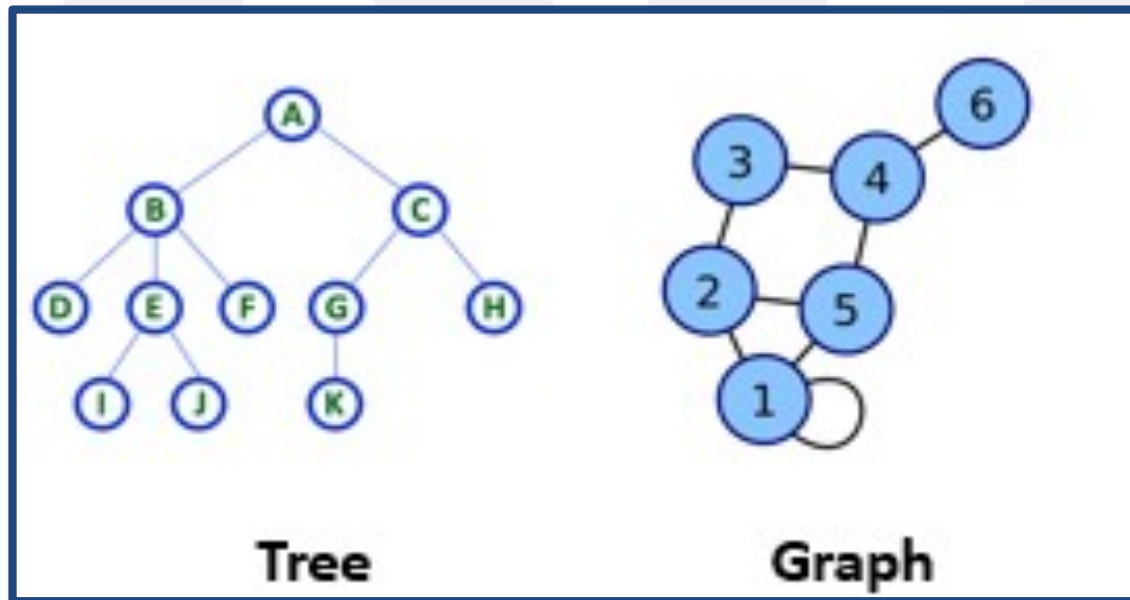
Non Primitive Data Structure : Linear data structures

- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.

  - Trees represent the hierarchical relationship between various elements.

  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Non Primitive Data Structure : Linear data structures

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

- A tree can be viewed as restricted graph.

- Graphs have many types:

Un-directed Graph   Multi Graph   Simple Graph

Directed Graph   Mixed Graph   Null Graph

      Weighted Graph

Difference between Linear and Non Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

Topic-3

Operation on Data structure

Operation on Data structure

- Operation means processing the data in the data structure. The following are some important operations

- **Create**
  - The create operation results in reserving memory for program elements.

- **Destroy**
  - Destroy operation destroys memory space allocated for specified data structure

- **Selection**
  - Selection operation deals with accessing a particular data within a data structure

Operation on Data structure

- **Traversing**
  - Traversal is a process of visiting each and every node of a list in systematic manner

- **Updation**
  - It updates or modifies the data in the data structure.

- **Searching**
  - To search for a particular value in the data structure for the given key value.

- **Sorting**
  - To arrange the values in the data structure in a particular order.

Operation on Data structure

- **Inserting**
  - To add a new value to the data structure

- **Deleting**
  - To remove a value from the data structure

- **Splitting**
  - Splitting is a process of partitioning single list to multiple list.

- **Merging**
  - To join two same type of data structure values

Parul® University

Topic-4

Review of an Array

Review of an Array

Overview of Arrays in C

An array is a collection of variables of the same type that are stored in contiguous memory locations. Arrays allow you to store multiple items of the same type together, which can be accessed using an index.

Declaration and Initialization

Declaration: You declare an array by specifying the type of its elements and the number of elements required by an array as follows:

```
datatype arrayName[arraySize];

int numbers[5]; // an array of 5 integers
```

Review of an Array

- **Initialization:** Arrays can be initialized at the time of declaration. For example:
  int numbers[5] = {1, 2, 3, 4, 5}; // an array of 5 integers with values

- If you don't specify all elements, the remaining will be initialized to 0.
  int numbers[5] = {1, 2}; // {1, 2, 0, 0, 0}

- **Accessing Elements**
- Array elements are accessed using their index. Indexing starts from 0, so the first element is arrayName[0].
  int first = numbers[0]; // Access the first element
  numbers[2] = 10;   // Set the third element to 10

Review of an Array

- **Iterating Through an Array**
  You can use loops to iterate through the elements of an array. A for
  loop is commonly used for this purpose.

```
for(int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
}
```

Multi-dimensional Arrays

C supports multi-dimensional arrays. For example, a two-dimensional array (a matrix) can be declared and initialized as follows:

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Review of an Array

Key Points

- **Memory Allocation:** Arrays have a fixed size, which is determined at compile-time. The size of an array cannot be altered at runtime.
- **Contiguous Memory:** Array elements are stored in contiguous memory locations, which allows for efficient indexing and iteration.
- **Bounds Checking:** C does not perform bounds checking. Accessing elements outside the array's range can result in undefined behavior.
- **Pointers and Arrays:** In many contexts, the name of an array is treated as a pointer to its first element. For example, numbers is equivalent to &numbers[0].

Topic-5

Structures, Self – Referential Structures, and Unions

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type.

Syntax

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
    };
```

```c
#include <stdio.h>
struct str1 {
    int i;
    char c;
    float f;
    char s[30];
};
void main() {
    struct str1 var1 = { 1, 'A', 1.00, "GeeksforGeeks" };
    printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",var1.i, var1.c, var1.f, var1.s);
    return 0;
}
```
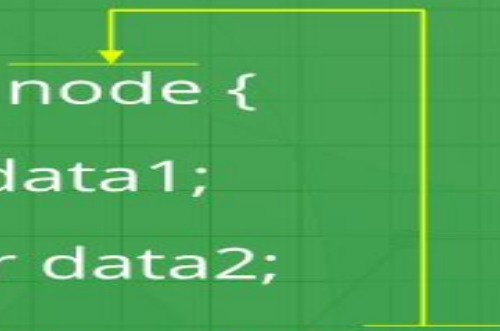
Self – Referential Structure

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Self Referential Structures

```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

Self – Referential Structure

```
struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob;
    return 0;
}
```

The Union is a user-defined data type in C language that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

Syntax of Union in C

union *union_name* {

   datatype member1;

   datatype member2;

      };

Union

Different Ways to Define a Union Variable

1. With Union Declaration
2. After Union Declaration

Defining Union Variable with Declaration

```
union union_name {
    datatype member1;
    datatype member2;
    ...
} var1, var2, ...;
```

Defining Union Variable after Declaration

```
union union_name var1,
var2, var3...;
```

Union

```c
#include <stdio.h>
// union template or declaration
union un {
    int member1;
    char member2;
    float member3;
};
```

```c
// driver code
void main()   {
    union un var1;
    var1.member1 = 15;
    printf("The value stored in member1 = %d", var1.member1);
}
```

Pointers and Dynamic Memory Allocation Functions

Pointer:

A pointer is a variable that stores the memory address of another variable.

Declaration and Initialization:

int *ptr;   // Declare a pointer to an integer

int var = 10;

ptr = &var;   // Initialize the pointer with the address of var

Dereferencing:

Access the value stored at the pointer's address using the * operator.

int value = *ptr; // value is now 10

Pointers and Dynamic Memory Allocation Functions

Key Points:

Null Pointer: Initialized to NULL to avoid undefined behavior.

Pointer to Pointer: Stores the address of another pointer.
    int **pptr = &ptr; // Pointer to a pointer

Use Cases: Efficient array manipulation, dynamic memory, complex data structures.

Pointers and Dynamic Memory Allocation Functions

Key Points:

Null Pointer: Initialized to NULL to avoid undefined behavior.

Pointer to Pointer: Stores the address of another pointer.
    int **pptr = &ptr; // Pointer to a pointer

Use Cases: Efficient array manipulation, dynamic memory, complex data structures.

Dynamic Memory Allocation Functions

C malloc() method

The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc() in C :   ptr = (cast-type*) malloc(byte-size)

For Example: *ptr = (int*) malloc(100 * sizeof(int));*

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.
And, the pointer ptr holds the address of the first byte in the allocated memory.

Dynamic Memory Allocation Functions

Malloc()

int* ptr = (int*) malloc (5* sizeof (int));;

4 bytes

ptr =

20 bytes of memory

A large 20 bytes memory block is dynamically allocated to ptr

Calloc()

int* ptr = (int*) calloc (5, sizeof (int));;

4 bytes

ptr =

4b

20 bytes of memory

5 blocks of 4 bytes each is dynamically allocated to ptr

Dynamic Memory Allocation Functions

C calloc() method

1. **"calloc"** or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

2. It initializes each block with a default value '0'.

3. It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C: ptr = (cast-type*)calloc(n, element-size);

For Example: *ptr = (float*) calloc(25, sizeof(float));*

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

Dynamic Memory Allocation Functions

C free() method

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C: free(ptr);
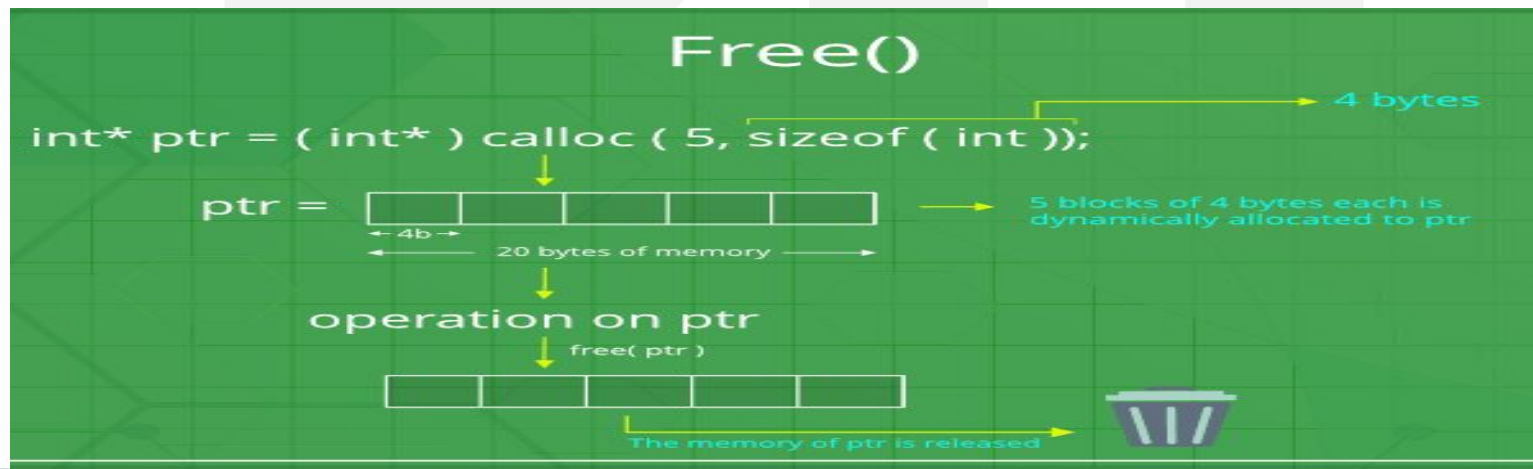
Dynamic Memory Allocation Functions

C realloc() method

"realloc" or **"re-allocation"** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.

Dynamic Memory Allocation Functions

Realloc()

4 bytes

int* ptr = (int*) malloc (5* sizeof (int));

ptr =

← 20 bytes of memory →

A large 20 bytes memory block is dynamically allocated to ptr

ptr = realloc (ptr, 10* sizeof(int));

ptr =

← 40 bytes of memory →

The size of ptr is changed from 20 bytes to 40 bytes dynamically

Topic – 6

Representation of Linear Arrays in Memory, dynamically allocated arrays

Representation of Linear Arrays in Memory, dynamically allocated arrays

This procedure is referred to as **Dynamic Memory Allocation in C**. Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Representation of Linear Arrays in Memory

Contiguous Allocation: Elements are stored in consecutive memory locations.

Address Calculation: Address of arr[i]=Base address+i×sizeof(type)

Example: For int arr[5] = {10, 20, 30, 40, 50} with a base address 0x1000:

Key Points

- **Efficient Access:** Direct via indices.
- **Fixed Size:** Set at compile time.
- **Memory Efficiency:** Contiguous storage for fast access.

Representation of Linear Arrays in Memory

Example

For int arr[5] = {10, 20, 30, 40, 50} with a base address 0x1000:

Index	Address	Value
arr[0]	0x1000	10
arr[1]	0x1004	20
arr[2]	0x1008	30
arr[3]	0x100C	40
arr[4]	0x1010	50

Accessing Elements

By index: int value = arr[2]; // 30

By pointer: int value = *(arr + 2); // 30

Representation of Linear Arrays in Memory

Dynamically allocated arrays: Allow flexible array sizes at runtime using pointers and memory functions.

Key Functions
1. malloc: Allocates memory.   int *arr = (int*)malloc(n * sizeof(int));
2. free: Frees allocated memory.   free(arr);

Key Points

Dynamic Size: Set at runtime.
Efficient Use: Allocate and free as needed.

Representation of Linear Arrays in Memory

Example:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int *arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

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

    free(arr);
    return 0;
}
```

Parul® University

Topic – 7

Analysis of an Algorithm

Algorithm

- Algorithms are the ideas behind computer programs.

- **What is algorithm? :** It is a finite set of instruction for performing a particular task.

- Data structures are implemented using algorithms.

The efficient algorithm

- There is always more than one algorithm to solve particular problem.

- Now there question out of many choices, which algorithm is to be used? Answer is, for that we need to analysis algorithms.

- To compare the performance of different algorithm and choose the best one to solve a particular problem, analysis of algorithm is needed.

The efficient algorithm

What is to be analyzed?

Algorithm Analysis Measures

1. Input size

2. Measuring Time complexity

3. Measuring Space complexity

4. Computing Best case, Average case, Worst case

5. Computing order of growth of algorithm.

1. Input size

- Input size depends on the problem being studied

- For many problems, such as **sorting** input size is the number of items in the input—for example, the array size n for sorting.

- If the input to an algorithm is a **graph**, the input size can be described by the numbers of vertices and edges in the graph

2. Time complexity

•The **amount of time** required by an algorithm to be executed is called its time complexity

•Time complexity is commonly estimated by counting the **number of elementary operations** performed on a particular input by the algorithm.

Time complexity- Example

- *Time complexity using frequency count*

```
for(i=0;i<n;i++)
{
printf("hello");
}
```

statement	count
i=0	1
i<n	n+1
i++	n
printf("hello")	n
Total =1+(n+1)+n+n=3n+2=O(n)	

Image source : Google

Time complexity- Example

• *Time complexity using frequency count*

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

Count++

}

}

statement	count
i=0	1
i<n	n+1
i++	n
J=0	n
j<n	n(n+1)
j++	n*n
Count++	n*n

Total =1+(n+1)+n+n+n(n+1)+n*n+n*n

=3n²+4n+2   =O(n²)

Image source : Google

3. Space complexity

•Space Complexity of an algorithm is total **space taken** by the algorithm with respect to the input size

•We often speak of **"extra" memory** needed, not counting the memory needed to store the input itself.

•Space complexity is sometimes **ignored** because the space used is minimal and/or obvious, but sometimes it becomes an important issue as time.

Space complexity: Which one is better?

```
largest = a
if b > largest then
    largest = b
end if
if c > largest then
    largest = c
end if
if d > largest then
    largest = d
end if
return largest
```

```
if a > b then
    if a > c then
        if a > d then
            return a
        else
            return d
        end if
    else
        if c > d then
            return c
        else
            return d
        end if
    end if
else
    if b > c then
        if b > d then
            return b
        else
            return d
        end if
    else
        if c > d then
            return c
        else
            return d
        end if
    end if
end if
```

Image source : Google

4. Computing Best case, Average case, Worst case

•The best, worst and average cases of a given algorithm express what the resource usage is **at least, at most and on average,** respectively

•The resource being considered is running time, but it could also be memory or the other resource

Computing Best case

- In the best case analysis, we calculate lower bound on running
time of an algorithm.

- We must know the case that causes minimum number of operations to be executed.

- In the linear search problem, the best case occurs when x is present at the first location.

Computing Average case

• In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.

• Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

• For the linear search problem we sum all the cases and divide the sum by (n)

Computing Worst case

- In the worst case analysis, we calculate upper bound on running time of an algorithm.

- We must know the case that causes maximum number of operations to be executed.

- e.g. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.

- When x is not present, the search () functions compares it with all the elements of array one by one

5. Order of growth of algorithm

- Order of growth in algorithm means how the time for computation increases when you increase the input size.

- It really matters when your input size is very large.
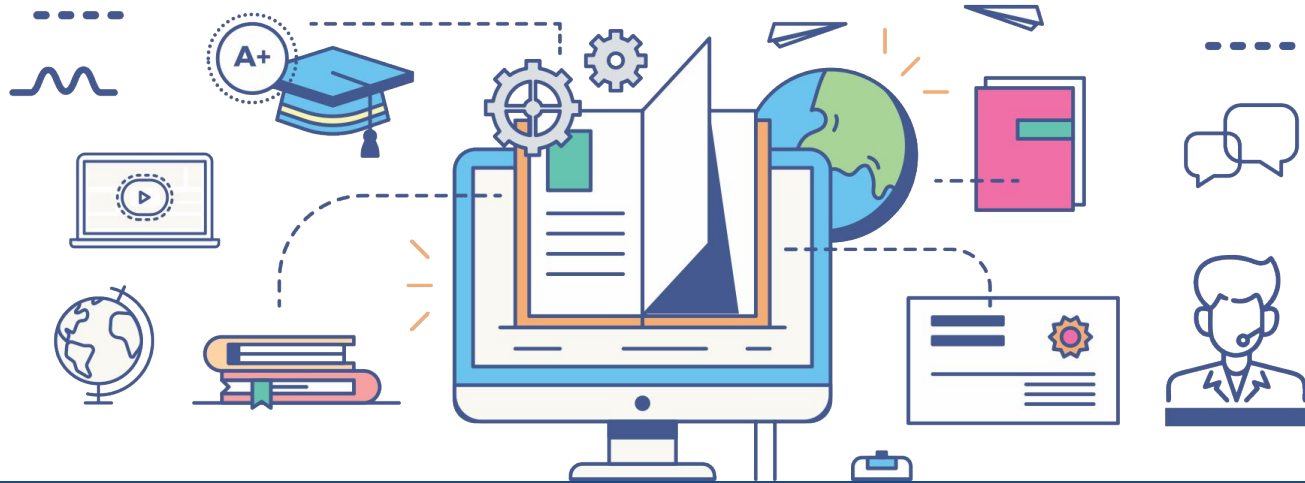
Order of growth of algorithm: example

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Image source : Google

Order of growth of algorithm:

Relationship between different rates of growth

Image source : Google
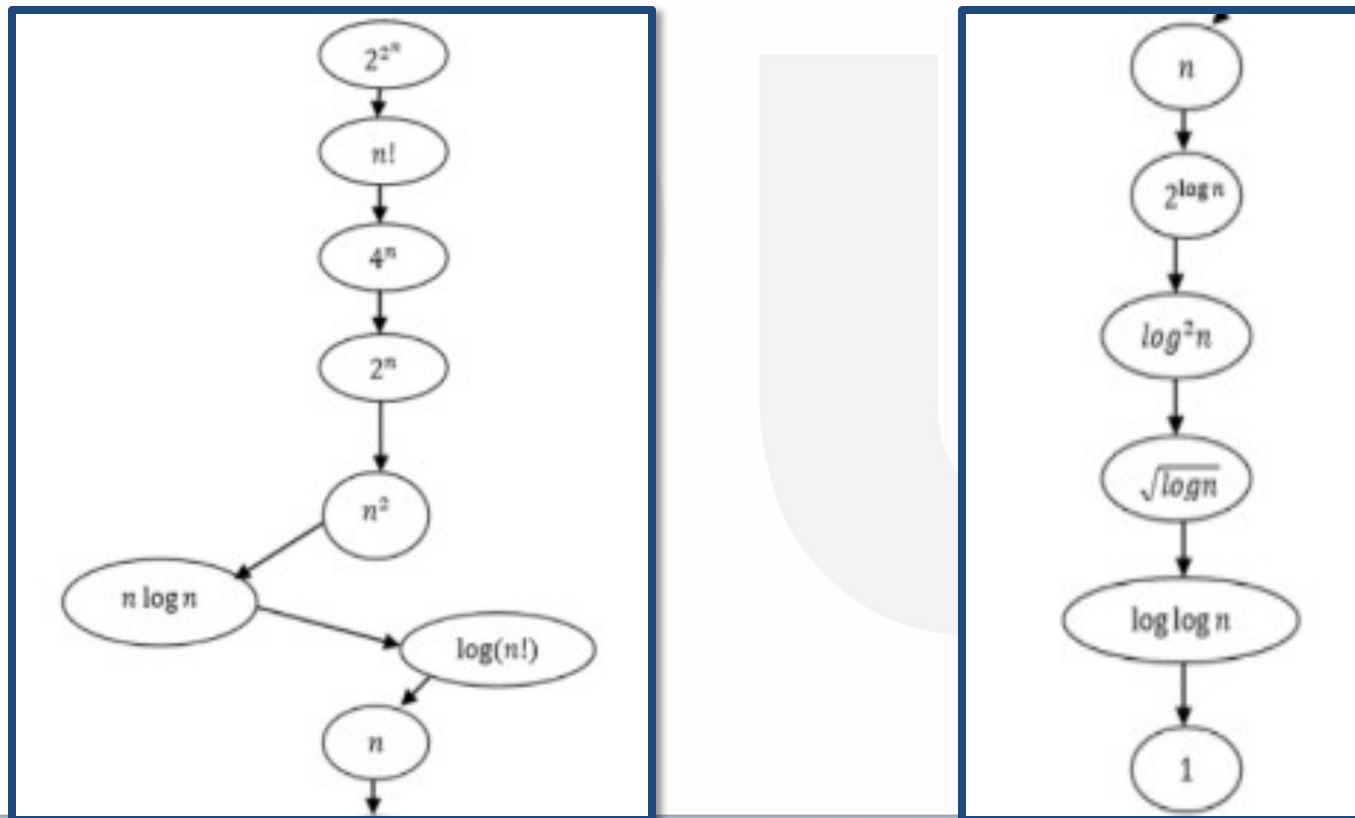
DIGITAL LEARNING CONTENT

Parul® University