

ATME College of Engineering, Mysuru

Department of CSE

Subject Presentation

1BCS305 – DATA STRUCTURES AND APPLICATIONS

Faculty: **Prof. Sandesh R**

Subject Code	1BCS305		CIE Marks	50
Number of Lecture / Hours	3		Exam Marks	50
Total No. of Lecture Hours	42		Exam Hours	03

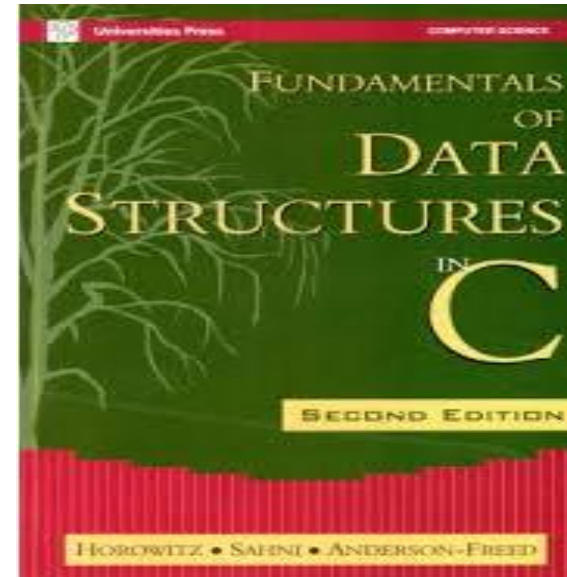
Course outcomes:

The students should be able to:

1. Recall the fundamental concepts, terminology, and characteristics of various data structures.
2. Explain the principles, representation, and operations of linear and non-linear data structures.
3. Implement different primitive data structures, their operations, and primitive applications.
4. Choose a suitable data structure to solve computational and real-world problems.
5. Make use of hashing techniques and leftist trees to organise the data effectively.

Text Books:

1. Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, **Fundamentals of Data Structures in C**, 2nd Ed, Universities Press, 2023



INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations **Review of pointers and dynamic Memory Allocation**

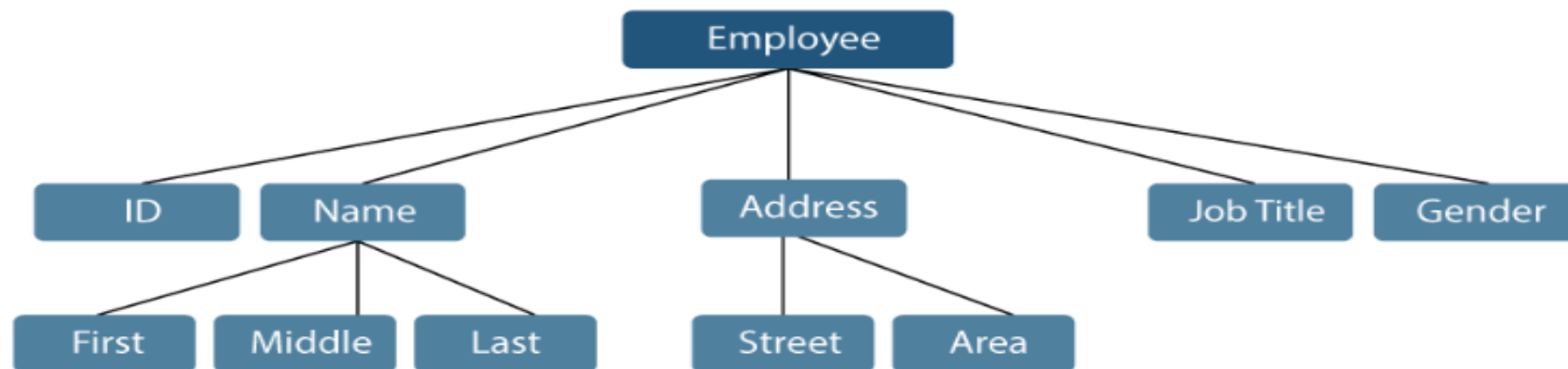
ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings

STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions

INTRODUCTION TO DATA STRUCTURES

Data is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values.

The **data items** are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.



INTRODUCTION TO DATA STRUCTURES

D A T A STRUCTURE

A data structure is a class of data that can be characterized by its Organization and the operations that are defined on it.

Data Structure = Organized Data + Allowed Operations

In other words, the organized collection of data is called data Structure A Data structure is a set of values along with the set of operations permitted on them It is also required to specify the semantics of operations permitted on the data values, and this is done by using set of axioms, which describes how these operations work.

INTRODUCTION TO DATA STRUCTURES

Basic Terminologies related to Data Structures

The following are some fundamental terminologies used whenever the data structures are involved:

Data: We can define data as an elementary value or a collection of values. For example, the Employee's name and ID are the data related to the Employee.

Data Items: A Single unit of value is known as Data Item.

Group Items: Data Items that have subordinate data items are known as Group Items. For example, an employee's name can have a first, middle, and last name.

Elementary Items: Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.

Entity and Attribute: A class of certain objects is represented by an Entity. It consists of different Attributes. Each Attribute symbolizes the specific property of that Entity.

INTRODUCTION TO DATA STRUCTURES

The term "**information**" is sometimes utilized for data with given attributes of meaningful or processed data.

Field: A single elementary unit of information symbolizing the Attribute of an Entity is known as Field.

Record: A collection of different data items are known as a Record. For example, if we talk about the employee entity, then its name, id, address, and job title can be grouped to form the record for the employee.

File: A collection of different Records of one entity type is known as a File. For example, if there are 100 employees, there will be 25 records in the related file containing data about each employee.

INTRODUCTION TO DATA STRUCTURES

Understanding the Need for Data Structures

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more.

Data Structures support different methods to organize, manage, and store data efficiently. With the help of Data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction.

Why should we learn Data Structures?

Data Structures and Algorithms are two of the key aspects of Computer Science.

Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.

Learning Data Structures and Algorithms will help us become better Programmers.

We will be able to write code that is more effective and reliable.

We will also be able to solve problems more quickly and efficiently.

INTRODUCTION TO DATA STRUCTURES

Understanding the Objectives of Data Structures

Data Structures satisfy two complementary objectives:

Correctness: Data Structures are designed to operate correctly for all kinds of inputs based on the domain of interest.

In other words, correctness forms the primary objective of Data Structure, which always depends upon the problems that the Data Structure is meant to solve.

Efficiency: Data Structures also require to be efficient. It should process the data quickly without utilizing many computer resources like memory space.

In a real-time state, the efficiency of a data structure is a key factor in determining the success and failure of the process.

INTRODUCTION TO DATA STRUCTURES

Understanding some Key Features of Data Structures

Some of the Significant Features of Data Structures are:

Robustness: Generally, all computer programmers aim to produce software that yields correct output for every possible input, along with efficient execution on all hardware platforms. This type of robust software must manage both valid and invalid inputs.

Adaptability: Building software applications like Web Browsers, Word Processors, and Internet Search Engine include huge software systems that require correct and efficient working or execution for many years. Moreover, software evolves due to emerging technologies or ever-changing market conditions.

Reusability: The features like Reusability and Adaptability go hand in hand. It is known that the programmer needs many resources to build any software, making it a costly enterprise. However, if the software is developed in a reusable and adaptable way, then it can be applied in most future applications. Thus, by executing quality data structures, it is possible to build reusable software, which appears to be cost-effective and timesaving.

INTRODUCTION TO DATA STRUCTURES

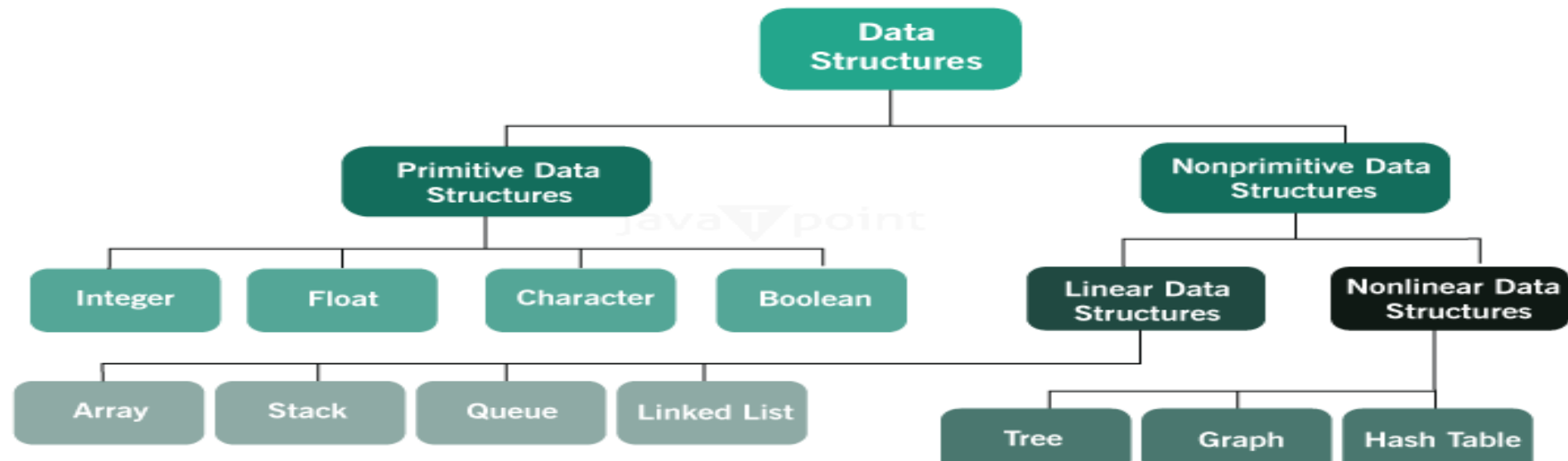
Classification of Data Structures

We can classify Data Structures into two categories:

Primitive Data Structure

Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.



INTRODUCTION TO DATA STRUCTURES

Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come in-built into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer**, **Float**, **Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

INTRODUCTION TO DATA STRUCTURES

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

Linear Data Structures

Non-Linear Data Structures

INTRODUCTION TO DATA STRUCTURES

Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a **Linear Data Structure**.

The arrangement of the data is done linearly, where each element consists of the **successors** and **predecessors** except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

INTRODUCTION TO DATA STRUCTURES

Based on memory allocation, the **Linear Data Structures** are further classified into two types:

Static Data Structures: The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.

The Array is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.

Dynamic Data Structures: The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.

Linked Lists, Stacks, and Queues are common examples of dynamic data structures

INTRODUCTION TO DATA STRUCTURES

Types of Linear Data Structures

1. Arrays

An Array is a data structure used to collect multiple data elements of the same data type into one variable.

Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable.

An Array is a list of elements where each element has a unique place in the list. The data elements of the array share the same variable name; however, each carries a different index number called a subscript.

We can access any data element from the list with the help of its location in the list.

Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

INTRODUCTION TO DATA STRUCTURES

Arrays can be classified into different types:

One-Dimensional Array: An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.

Two-Dimensional Array: An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.

Multidimensional Array: We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices as per the need.

INTRODUCTION TO DATA STRUCTURES

Some Applications of Array:

We can store a list of data elements belonging to the same data type.

Array acts as an auxiliary storage for other data structures.

The array also helps store data elements of a binary tree of the fixed count.

Array also acts as a storage of matrices.

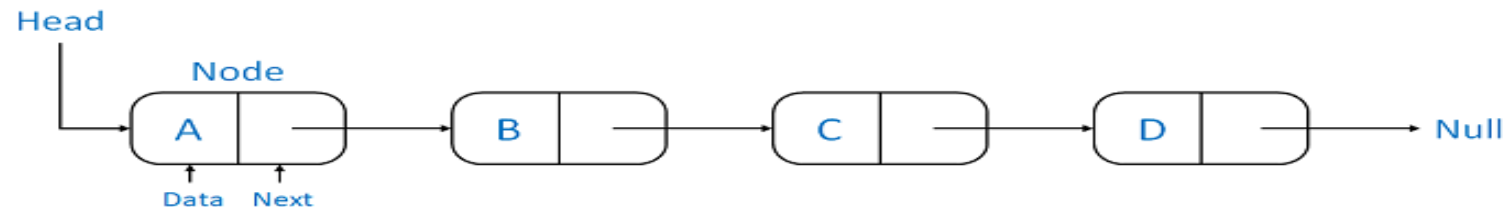
INTRODUCTION TO DATA STRUCTURES

2. Linked Lists

A Linked List is another example of a linear data structure used to store a collection of data elements dynamically.

Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list.

The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.



INTRODUCTION TO DATA STRUCTURES

Linked Lists can be classified into different types:

Singly Linked List: A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

Doubly Linked List: A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).

Circular Linked List: The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

INTRODUCTION TO DATA STRUCTURES

Some Applications of Linked Lists:

The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.

We can also implement Operating System's function for dynamic memory management.

Linked Lists also allow polynomial implementation for mathematical operations.

We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.

Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.

Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

INTRODUCTION TO DATA STRUCTURES

3. Stacks

A Stack is a Linear Data Structure that follows the LIFO (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top.

Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

The figure represents the real-life example of a Stack where the operations are performed from one end only, like the insertion and removal of new books from the top of the Stack. It implies that the insertion and deletion in the Stack can be done only from the top of the Stack. We can access only the Stack's tops at any given time.

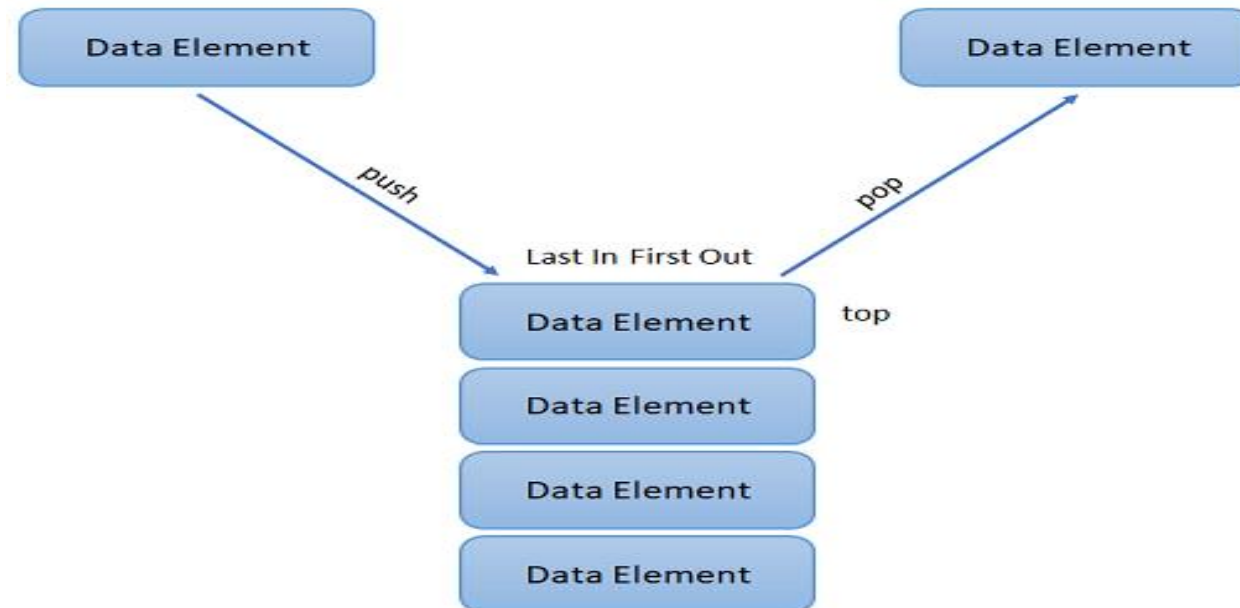


INTRODUCTION TO DATA STRUCTURES

The primary operations in the Stack are as follows:

Push: Operation to insert a new element in the Stack is termed as Push Operation.

Pop: Operation to remove or delete elements from the Stack is termed as Pop Operation.



INTRODUCTION TO DATA STRUCTURES

Some Applications of Stacks:

The Stack is used as a Temporary Storage Structure for recursive operations.

Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.

We can manage function calls using Stacks.

Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.

Stacks are also helpful in converting infix expressions to postfix expressions.

INTRODUCTION TO DATA STRUCTURES

Some Applications of Stacks:

Stacks allow us to check the expression's syntax in the programming environment.

We can match parenthesis using Stacks.

Stacks can be used to reverse a String.

Stacks are helpful in solving problems based on backtracking.

We can use Stacks in depth-first search in graph and tree traversal.

Stacks are also used in Operating System functions.

Stacks are also used in UNDO and REDO functions in an edit.

INTRODUCTION TO DATA STRUCTURES

4. Queues

A Queue is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows **FIFO (First In, First Out)** principle to manipulate the data elements.

Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

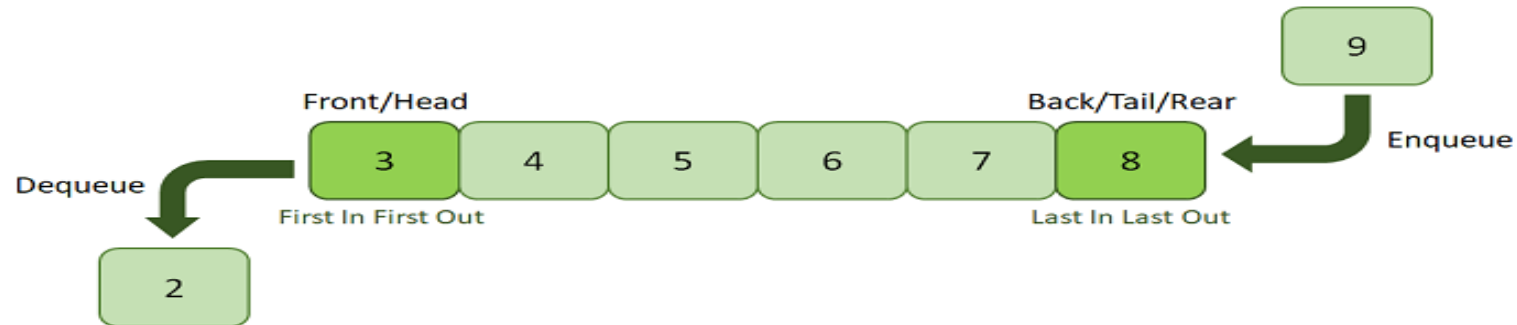


INTRODUCTION TO DATA STRUCTURES

The following are the primary operations of the Queue:

Enqueue: The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.

Dequeue: Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.



INTRODUCTION TO DATA STRUCTURES

Some Applications of Queues:

Queues are generally used in the breadth search operation in Graphs.

Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.

Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.

Priority Queues are utilized in file-downloading operations in a browser.

Queues are also used to transfer data between peripheral devices and the CPU.

Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

INTRODUCTION TO DATA STRUCTURES

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order.

Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

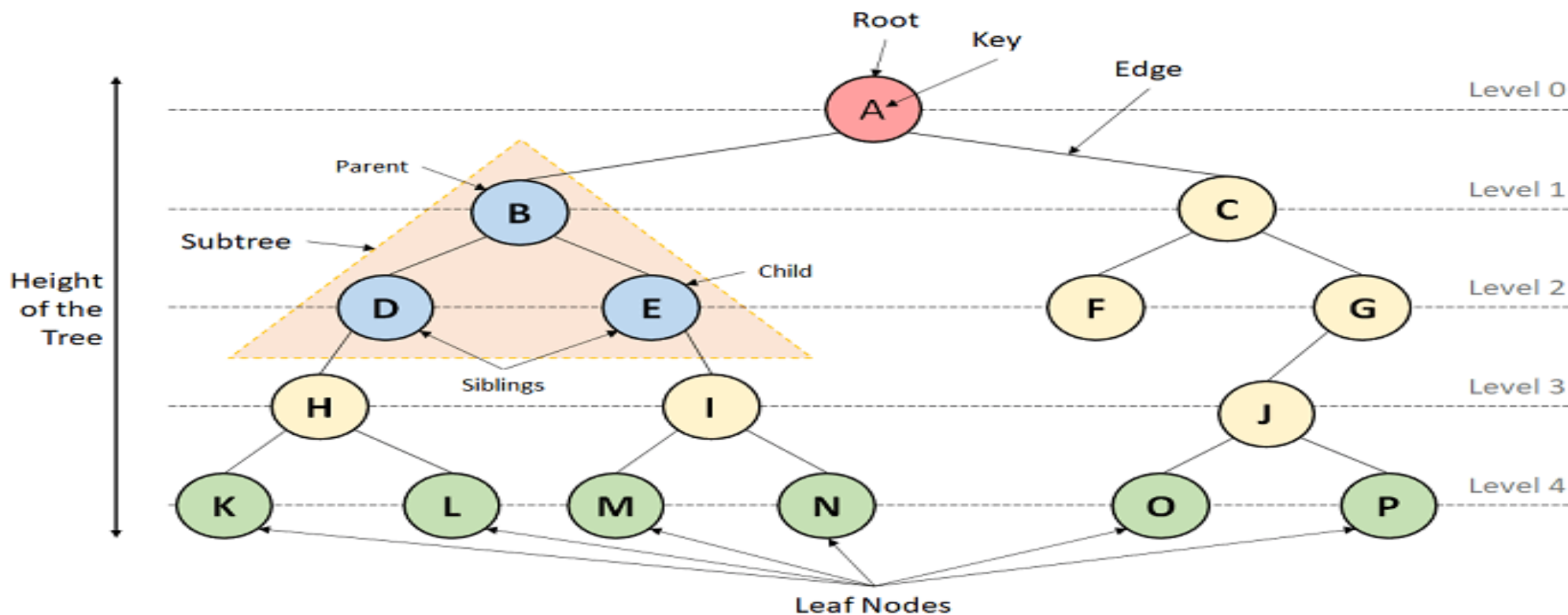
Types of Non-Linear Data Structures

1. Trees

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively.

It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



INTRODUCTION TO DATA STRUCTURES

Trees can be classified into different types:

Binary Tree: A Tree data structure where each parent node can have at most two children is termed a Binary Tree.

Binary Search Tree: A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

AVL Tree: An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.

B-Tree: A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

INTRODUCTION TO DATA STRUCTURES

Some Applications of Trees:

Trees implement hierarchical structures in computer systems like directories and file systems.

Trees are also used to implement the navigation structure of a website.

Trees are also helpful in decision-making in Gaming applications.

Trees are responsible for implementing priority queues for priority-based OS scheduling functions.

Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.

We can use Trees to store data keys for indexing for Database Management System (DBMS).

Spanning Trees allows us to route decisions in Computer and Communications Networks.

Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

INTRODUCTION TO DATA STRUCTURES

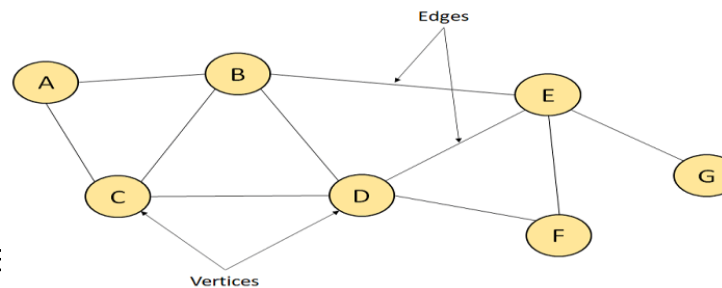
2. Graphs

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them.

The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks.

For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below: $G = (V, E)$



The above figure represents a Graph having ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

and ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

INTRODUCTION TO DATA STRUCTURES

Some Applications of Graphs:

Graphs help us represent routes and networks in transportation, travel, and communication applications.

Graphs are used to display routes in GPS.

Graphs also help us represent the interconnections in social networks and other network-based applications.

Graphs are utilized in mapping applications.

Graphs are responsible for the representation of user preference in e-commerce applications.

Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.

Graphs also help to manage the utilization and availability of resources in an organization.

Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.

Graphs are also used in robotic motions and neural networks.

INTRODUCTION TO DATA STRUCTURES

Basic Operations of Data Structures

Traversal: Traversing a data structure means accessing each data element exactly once so it can be administered.

For example, traversing is required while printing the names of all the employees in a department.

Search: Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints.

Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.

Insertion: Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.

INTRODUCTION TO DATA STRUCTURES

Deletion: Deletion means to remove or delete a specific data element from the given list of data elements.

For example, we can use the deleting operation to delete the name of an employee who has left the job.

Sorting: Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application.

For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.

Merge: Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.

INTRODUCTION TO DATA STRUCTURES

Create: Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:

- Compile-time

- Run-time

For example, the malloc() function is used in C Language to create data structure.

Selection: Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.

Update: The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.

Splitting: The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte. Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;
```

```
int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.
```

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol).

```
int *a;//pointer to int
```

```
char *c;//pointer to char
```

Let's see the pointer example

```
#include<stdio.h>
int main()
{
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives
the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we
print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

```
int arr[10];  
int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
void show (int);  
void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
struct st  
{  
    int i;  
    float f;  
}ref;  
struct st *p = &ref;
```

Advantage of pointer

1. Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
2. We can return multiple values from a function using the pointer.
3. It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

C Pointers

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>
int main()
{
int number=50;
printf("value of number is %d, address of number is %u", number, &number);
return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>
int main()
{
int a=10,b=20,*p1=&a,*p2=&b;
printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\n After swap: *p1=%d *p2=%d",*p1,*p2);
return 0;
}
```

Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

C Double Pointer (Pointer to Pointer)

However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer).

The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer.



The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

C Pointers

Consider the following example.

```
#include<stdio.h>
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed
    printf("address of p: %x\n",pp); // Address of p will be printed
    printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stored at pp
}
```

Output

```
address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10
```

Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory.

It allows different variables to be accessed by using a single pointer to the structure.

Advantages

It can hold variables of different data types.

We can create objects containing different types of attributes.

It allows us to re-use the data layout across programs.

It is used to implement other data structures like linked lists, stacks, queues, trees, graphs **etc**

Syntax

```
struct structure_name  
{  
    data_type member1;  
    data_type member2;  
    .  
    .  
    data_type member;  
};
```

Dynamic memory allocation in C

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of **stdlib.h** header file.

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

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Dynamic memory allocation in C

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Dynamic memory allocation in C

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Let's see the example of malloc() function:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
```

Dynamic memory allocation in C

```
if(ptr==NULL)
{
    printf("Sorry! unable to allocate memory");
    exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

Dynamic memory allocation in C

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
```

Dynamic memory allocation in C

```
if(ptr==NULL)
{
    printf("Sorry! unable to allocate memory");
    exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

Dynamic memory allocation in C

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

ptr=realloc(ptr, new-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

free(ptr)

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Linear Array

A linear array is a list of a finite number 'n' of homogeneous data element such that

- The elements of the array are reference respectively by an index set consisting of n consecutive numbers.
- The element of the array are respectively in successive memory locations.

The number n of elements is called the length or size of the array. The length or the numbers of elements of the array can be obtained from the index set by the formula

When $LB = 0$, **Length = $UB - LB + 1$**

When $LB = 1$, **Length = UB**

Where, UB is the largest index called the **Upper Bound**

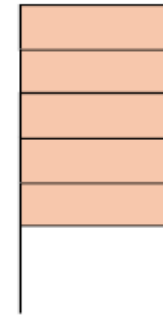
LB is the smallest index, called the **Lower Bound**

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Representation of linear arrays in memory

Let LA be a linear array in the memory of the computer. The memory of the computer is simply a sequence of address location as shown below,

1000
1001
1002
1003
1004



$$\text{LOC (LA [K])} = \text{address of the element LA [K] of the array LA}$$

The elements of LA are stored in successive memory cells.

The computer does not keep track of the address of every element of LA, but needs to keep track only the address of the first element of LA denoted by, **Base (LA)**

Using the base address of LA, the computer calculates the address of any element of LA by the formula

$$\text{LOC (LA[K])} = \text{Base(LA)} + w(K - \text{lower bound})$$

One Dimensional Array

While writing computer programs, if finds ourselves in a situation where we cannot determine how large an array to use, then a good solution to this problem is to defer this decision to run time and allocate the array when we have a good estimate of the required array size.

Example:

```
int i, n, *list;
printf("Enter the number of numbers to generate:");
scanf("%d", &n);
if(n<1)
{
    fprintf (stderr, "Improper value of n \n");
    exit(EXIT_FAILURE);
}
MALLOC (list, n*sizeof(int));
```

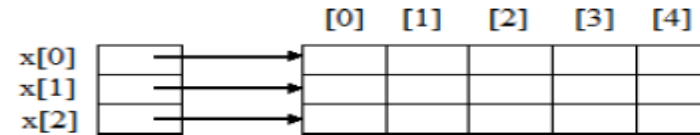
The programs fails only when $n < 1$ or insufficient memory to hold the list of numbers that are to be sorted.

DYNAMICALLY ALLOCATED ARRAYS

Two Dimensional Arrays

C uses array-of-arrays representation to represent a multidimensional array. The two dimensional arrays is represented as a one-dimensional array in which each element is itself a one-dimensional array.

Example: `int x[3][5];`



Array-of-arrays representation

C find element `x[i][j]` by first accessing the pointer in `x[i]`.

Where `x[i] =  $\alpha + i * \text{sizeof}(\text{int})$` , which give the address of the zeroth element of row `i` of the array.

Then adding `j* $\text{sizeof}(\text{int})$` to this pointer (`x[i]`), the address of the `[j]`th element of row `i` is determined.

$$x[i] = \alpha + i * \text{sizeof}(\text{int})$$

$$x[j] = \alpha + j * \text{sizeof}(\text{int})$$

$$x[i][j] = x[i] + j * \text{sizeof}(\text{int})$$

Creation of Two-Dimensional Array Dynamically

```
int **myArray;  
myArray = make2dArray(5,10);  
myArray[2][4]=6;  
  
int ** make2dArray(int rows, int cols)  
{ /* create a two dimensional rows X cols array */  
    int **x, i;  
    MALLOC(x, rows * sizeof (*x)); /*get memory for row pointers*/  
    for (i= 0;i<rows; i++)           /* get memory for each row */  
        MALLOC(x[i], cols *sizeof(**x));  
    return x;  
}
```

The second line allocates memory for a 5 by 10 two-dimensional array of integers and the third line assigns the value 6 to the [2][4] element of this array.

1. Traversing

Let A be a collection of data elements stored in the memory of the computer. Suppose if the contents of the each elements of array A needs to be printed or to count the numbers of elements of A with a given property can be accomplished by Traversing.

Traversing is a accessing and processing each element in the array exactly once.

Algorithm 1: (Traversing a Linear Array)

Let LA is a linear array with the lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA using while loop.

- | | |
|-------------------------|-------------------------|
| 1. [Initialize Counter] | set $K := LB$ |
| 2. Repeat step 3 and 4 | while $K \leq UB$ |
| 3. [Visit element] | Apply PROCESS to LA [K] |
| 4. [Increase counter] | Set $K := K + 1$ |
| [End of step 2 loop] | |
| 5. Exit | |

ARRAY OPERATIONS



2. Inserting

Let A be a collection of data elements stored in the memory of the computer. Inserting refers to the operation of adding another element to the collection A.

Inserting an element at the “end” of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.

Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the other elements.

Algorithm:

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the K^{th} position in LA.

- | | |
|--|-----------------------|
| 1. [Initialize counter] | set J:= N |
| 2. Repeat step 3 and 4 | while J ≥ K |
| 3. [Move J th element downward] | Set LA [J+1] := LA[J] |
| 4. [Decrease counter] | set J:= J – 1 |
| [End of step 2 loop] | |
| 5. [Insert element] | set LA[K]:= ITEM |
| 6. [Reset N] | set N:= N+1 |
| 7. Exit | |

3. Deleting

Deleting refers to the operation of removing one element to the collection A.

Deleting an element at the “end” of the linear array can be easily done with difficulties.

If element at the middle of the array needs to be deleted, then each subsequent elements be moved one location upward to fill up the array.

Algorithm

DELETE (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. this algorithm deletes the K^{th} element from LA

1. Set $\text{ITEM} := \text{LA}[K]$
2. Repeat for $J = K$ to $N - 1$
 [Move $J + 1$ element upward] set $\text{LA}[J] := \text{LA}[J+1]$
 [End of loop]
3. [Reset the number N of elements in LA] set $N := N - 1$
4. Exit

4. Sorting

Sorting refers to the operation of rearranging the elements of a list. Here list be a set of n elements. The elements are arranged in increasing or decreasing order.

Ex: suppose A is the list of n numbers. Sorting A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that,

$$A[1] < A[2] < A[3] < \dots < A[N]$$

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

5. Searching

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. Searching refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there.

The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.

Linear Search

Algorithm: (Linear Search) LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC: = 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA [N + 1]: = ITEM.
2. [Initialize counter.] Set LOC: = 1.
3. [Search for ITEM.]
Repeat while| DATA [LOC] $\neq$ ITEM:
Set LOC:= LOC + 1.
[End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC:=0
5. Exit.

Binary Search

Algorithm: (Binary Search) **BINARY (DATA, LB, UB, ITEM, LOC)**

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, the beginning, end and middle locations of a segment of elements of DATA.

This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM.
3. If ITEM < DATA [MID], then:
Set END := MID - 1.
Else:
Set BEG := MID + 1.
[End of If structure.]
4. Set MID := INT((BEG + END)/2).
[End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
Set LOC := MID.
Else:
Set LOC := NULL.
[End of If structure.]
6. Exit.

C Program to Calculate Average Using Arrays

```
#include <stdio.h>
int main() {
    int n, i;
    float num[100], sum = 0.0, avg;

    printf("Enter the numbers of elements: ");
    scanf("%d", &n);

    while (n > 100 || n < 1) {
        printf("Error! number should in range of (1 to 100).\n");
        printf("Enter the number again: ");
        scanf("%d", &n);
    }
```

C Program to Calculate Average Using Arrays

```
for (i = 0; i < n; ++i) {  
    printf("%d. Enter number: ", i + 1);  
    scanf("%f", &num[i]);  
    sum += num[i];  
}  
  
avg = sum / n;  
printf("Average = %.2f", avg);  
return 0;  
}
```

Output

```
Enter the numbers of elements: 6  
1. Enter number: 45.3  
2. Enter number: 67.5  
3. Enter number: -45.6  
4. Enter number: 20.34  
5. Enter number: 33  
6. Enter number: 45.6  
Average = 27.69
```

C Program to Find Largest Element in an Array

```
#include <stdio.h>
int main() {
    int n;
    double arr[100];
    printf("Enter the number of elements (1 to 100): ");
    scanf("%d", &n);

    for (int i = 0; i < n; ++i) {
        printf("Enter number%d: ", i + 1);
        scanf("%lf", &arr[i]);
    }
```

C Program to Find Largest Element in an Array

```
// storing the largest number to arr[0]
for (int i = 1; i < n; ++i) {
    if (arr[0] < arr[i]) {
        arr[0] = arr[i];
    }
}

printf("Largest element = %.2lf", arr[0]);

return 0;
}
```

Output

```
Enter the number of elements (1 to 100): 5
Enter number1: 34.5
Enter number2: 2.4
Enter number3: -35.5
Enter number4: 38.7
Enter number5: 24.5
Largest element = 38.70
```

C Program to Find Largest Number Using Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    double *data;
    printf("Enter the total number of elements: ");
    scanf("%d", &n);

    // Allocating memory for n elements
    data = (double *)calloc(n, sizeof(double));
    if (data == NULL) {
        printf("Error!!! memory not allocated.");
        exit(0);
    }
}
```

C Program to Find Largest Number Using Dynamic Memory Allocation

```
// Storing numbers entered by the user.  
for (int i = 0; i < n; ++i) {  
    printf("Enter number%d: ", i + 1);  
    scanf("%lf", data + i);  
}  
/ Finding the largest number  
for (int i = 1; i < n; ++i) {  
    if (*data < *(data + i)) {  
        *data = *(data + i);  
    }  
}  
printf("Largest number = %.2lf", *data);  
free(data);  
return 0;  
}
```

Output

```
Enter the total number of elements: 5  
Enter number1: 3.4  
Enter number2: 2.4  
Enter number3: -5  
Enter number4: 24.2  
Enter number5: 6.7  
Largest number = 24.20
```

MULTIDIMENSIONAL ARRAY

Two-Dimensional Arrays

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two dimensional Array in C

The syntax to declare the 2D array is given below,

data_type array_name[rows][columns];

Consider the following example.

int twodimen[4][3];

Here, 4 is the number of rows, and 3 is the number of columns.

MULTIDIMENSIONAL ARRAY

Initialization of 2D Array in C

```
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

```
#include<stdio.h>
int main()
{
    int i=0,j=0;
    int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}}; //traversing 2D array
    for(i=0;i<4;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
        }
    }
    return 0;
}
```

Output

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

MULTIDIMENSIONAL ARRAY

C Program to Add Two Matrices Using Multi-dimensional Arrays

```
#include <stdio.h>
int main() {
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;
    printf("Enter the number of rows (between 1 and 100): ");
    scanf("%d", &r);
    printf("Enter the number of columns (between 1 and 100): ");
    scanf("%d", &c);

    printf("\nEnter elements of 1st matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }
```

MULTIDIMENSIONAL ARRAY

C Program to Add Two Matrices Using Multi-dimensional Arrays

```
printf("Enter elements of 2nd matrix:\n");
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        printf("Enter element b%d%d: ", i + 1, j + 1);
        scanf("%d", &b[i][j]);
    }

// adding two matrices
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        sum[i][j] = a[i][j] + b[i][j];
    }
```

MULTIDIMENSIONAL ARRAY

C Program to Add Two Matrices Using Multi-dimensional Arrays

```
// printing the result
printf("\nSum of two matrices: \n");
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        printf("%d ", sum[i][j]);
        if (j == c - 1) {
            printf("\n\n");
        }
    }

return 0;
}
```

Output

```
Enter the number of rows (between 1 and 100): 2
Enter the number of columns (between 1 and 100): 3

Enter elements of 1st matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 2
Enter element a23: 3
Enter elements of 2nd matrix:
Enter element b11: -4
Enter element b12: 5
Enter element b13: 3
Enter element b21: 5
Enter element b22: 6
Enter element b23: 3

Sum of two matrices:
-2  8  7

10  8  6
```

What is a polynomial?

“A polynomial is a sum of terms, where each term has a form ax^e , where x is the variable, a is the coefficient and e is the exponent.”

Two example polynomials are:

$$A(x) = 3x^{20} + 2x^5 + 4$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

The largest (or leading) exponent of a polynomial is called its **degree**.

Coefficients that are zero are not displayed. The term with exponent equal to zero does not show the variable since x raised to a power of zero is 1.

Assume there are two polynomials,

$$A(x) = \sum a_i x_i \text{ and } B(x) = \sum b_i x_i$$

then:

$$A(x) + B(x) = \sum (a_i + b_i) x_i$$

$$A(x) \cdot B(x) = \sum (a_i x_i \cdot \sum (b_j x_j))$$

Polynomial Representation

One way to represent polynomials in C is to use **typedef** to create the type polynomial as below:

```
#define MAX-DEGREE 101          /*Max degree of polynomial+1*/  
typedef struct{  
    int degree;  
    float coef[MAX-DEGREE];  
} polynomial;
```

Now if a is a variable and is of type polynomial and $n < \text{MAX_DEGREE}$, the polynomial $A(x) = \sum a_i x_i$ would be represented as:

```
a.degree = n  
a.coef[i] = an-i,  $0 \leq i \leq n$ 
```

Consider the two polynomials

$$A(x) = 2x^{1000} + 1$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

	startA	finishA	startB		finishB	avail
	↓	↓	↓		↓	↓
coef	2	1	1	10	3	1
exp	1000	0	4	3	2	0
	0	1	2	3	4	5
						6

The above figure shows how these polynomials are stored in the array terms. The index of the first term of A and B is given by startA and startB, while finishA and finishB give the index of the last term of A and B.

The index of the next free location in the array is given by avail.

For above example, **startA=0**, **finishA=1**, **startB=2**, **finishB=5**, & **avail=6**.

C Program on Additional of Two Polynomials

```
#include<stdio.h>
/* declare structure for polynomial */
struct poly {
    int coeff;
    int expo;
};

struct poly p1[10], p2[10], p3[10];

/* function prototypes */
int readPoly(struct poly[]);
int addPoly(struct poly[], struct poly[], int, int, struct poly[]);
void displayPoly(struct poly[], int terms);
```

POLYNOMIALS

```
int main(){
    int p1, p2, t1, t2, p3;
    /* read and display first polynomial */
    t1 = readPoly(p1);
    printf(" \n First polynomial : ");
    displayPoly(p1, t1);
    /* read and display second polynomial */
    t2 = readPoly(p2);
    printf(" \n Second polynomial : ");
    displayPoly(p2, t2);
    /* add two polynomials and display resultant polynomial */
    t3 = addPoly(p1, p2, t1, t2, p3);
    printf(" \n\n Resultant polynomial after addition : ");
    displayPoly(p3, t3);
    printf("\n");
    return 0;
}
```

```
int readPoly(struct poly p[10])
```

```
{
```

```
    int t1, i;
```

```
    printf("\n\n Enter the total number of terms in the polynomial:");
```

```
    scanf("%d", &t1);
```

```
    printf("\n Enter the COEFFICIENT and EXPONENT in DESCENDING ORDER\n");
```

```
    for (i = 0; i < t1; i++) {
```

```
        printf("  Enter the Coefficient(%d): ", i + 1);
```

```
        scanf("%d", &p[i].coeff);
```

```
        printf("    Enter the exponent(%d): ", i + 1);
```

```
        scanf("%d", &p[i].expo); /* only statement in loop */
```

```
    }
```

```
    return (t1);
```

```
}
```

```
int addPoly(struct poly p1[10], struct poly p2[10], int t1, int t2, struct poly p3[10])
```

```
{
    int i, j, k;
    i = 0;
    j = 0;
    k = 0;
    while (i < t1 && j < t2) {
        if (p1[i].expo == p2[j].expo) {
            p3[k].coeff = p1[i].coeff + p2[j].coeff;
            p3[k].expo = p1[i].expo;
            i++;
            j++;
            k++;
        }
        else if (p1[i].expo > p2[j].expo) {
            p3[k].coeff = p1[i].coeff;
            p3[k].expo = p1[i].expo;
            i++;
            k++;
        }
    }
}
```

```
else  
{  
    p3[k].coeff = p2[j].coeff;  
    p3[k].expo = p2[j].expo;  
    j++;  
    k++;  
}  
}
```

```
/* for rest over terms of polynomial 1 */  
while (i < t1) {  
    p3[k].coeff = p1[i].coeff;  
    p3[k].expo = p1[i].expo;  
    i++;  
    k++;  
}
```

// for rest over terms of polynomial 2 */

```
while (i < t2) {
    p3[k].coeff = p2[j].coeff;
    p3[k].expo = p2[j].expo;
    j++;
    k++;
}
return (k); /* k is number of terms in resultant polynomial*/
}
void displayPoly(struct poly p[10], int term)
{
    int k;

    for (k = 0; k < term - 1; k++)
        printf("%d(x^%d)+", p[k].coeff, p[k].expo);
    printf("%d(x^%d)", p[term - 1].coeff, p[term - 1].expo);
}
```



A T M E

College of Engineering

POLYNOMIALS



SPARSE MATRICES

A matrix contains m rows and n columns of elements as illustrated in below figures. In this figure, the

The first matrix has five rows and three columns and the second has six rows and six columns. We write m x n (read "m by n") to designate a matrix with m rows and n columns.

The total number of elements in such a matrix is mn. If m equals n, the matrix is square.

	col0	col1	col2
row 0	-27	3	4
row 1	6	82	-2
row 2	109	-64	11
row 3	12	8	9
row 4	48	27	47

Figure A

	col0	col1	col2	col3	col4	col 5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Figure B

A sparse matrix can be characterized by using the triple **<row,col,value>**. This means that, an array of triples is used to represent a sparse matrix.

Organize the triples so that the row indices are in ascending order.

The operations should terminate, so we must know the number of rows and columns, and the number of nonzero elements in the matrix.

Implementation of the **Create** operation as below:

SparseMatrix Create(maxRow, maxCol) ::=

```
#define MAX_TERMS 101          /* maximum number of terms +1 */
typedef struct {
    int    col;
    int    row;
    int value;
} term;
term a[MAX_TERMS];
```

C Program to Find Transpose of a Matrix

```
#include <stdio.h>
int main() {
    int a[10][10], transpose[10][10], r, c;
    printf("Enter rows and columns: ");
    scanf("%d %d", &r, &c);

    // assigning elements to the matrix
    printf("\nEnter matrix elements:\n");
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }
```

C Program to Find Transpose of a Matrix

```
// printing the matrix a[][]
printf("\nEntered matrix: \n");
for (int i = 0; i < r; ++i)
for (int j = 0; j < c; ++j) {
    printf("%d ", a[i][j]);
    if (j == c - 1)
        printf("\n");
}
```

```
// computing the transpose
for (int i = 0; i < r; ++i)
for (int j = 0; j < c; ++j) {
    transpose[j][i] = a[i][j];
}
```

C Program to Find Transpose of a Matrix

```
// printing the transpose
printf("\nTranspose of the matrix:\n");
for (int i = 0; i < c; ++i)
for (int j = 0; j < r; ++j) {
    printf("%d ", transpose[i][j]);
    if (j == r - 1)
        printf("\n");
}
return 0;
}
```

Output

```
Enter rows and columns: 2
3

Enter matrix elements:
Enter element a11: 1
Enter element a12: 4
Enter element a13: 0
Enter element a21: -5
Enter element a22: 2
Enter element a23: 7

Entered matrix:
1  4  0
-5  2  7

Transpose of the matrix:
1  -5
4  2
0  7
```

The string can be defined as the one-dimensional array of characters terminated by a **null ('\\0')**.

The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be **0**.

The termination character (**'\\0'**) is important in a string since it is the only way to identify where the string ends.

When we define a string as **char s[10]**, the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

```
char ch[10]={'a', 't', 'm', 'e', 'c', 'o', 'l', 'l', 'e', 'g', 'e', '\0'};
```

We can also define the string by the string literal in C language. For example: **char ch[]**="atme college";

In such case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal

There are two main differences between char array and literal.

We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.

The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

```
#include<stdio.h>
#include <string.h>
int main()
{
    char ch[11]={'a', 't', 'm', 'e', 'c', 'o', 'l', 'l', 'e', 'g', 'e', '\0'};
    char ch2[11]="atme college";
    printf("Char Array Value is: %s\n", ch);
    printf("String Literal Value is: %s\n", ch2);
    return 0;
}
```

Output:

Char Array Value is: atmecollege
String Literal Value is: atmecollege

Accepting string as the input

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%s",s);
    printf("You entered %s",s);
}
```

Output:

Enter the string? **Atme College**
You entered **Atme College**

C String Functions

No.	Function	Description
1)	<code>strlen(string_name)</code>	returns the length of string name.
2)	<code>strcpy(destination, source)</code>	copies the contents of source string to destination string.
3)	<code>strcat(first_string, second_string)</code>	concatenates or joins first string with second string. The result of the string is stored in first string.
4)	<code>strcmp(first_string, second_string)</code>	compares the first string with second string. If both strings are same, it returns 0.
5)	<code>strrev(string)</code>	returns reverse string.
6)	<code>strlwr(string)</code>	returns string characters in lowercase.
7)	<code>strupr(string)</code>	returns string characters in uppercase.

C String Functions

```
#include<stdio.h>
void main()
{
    char string1[25],string2[25];
    int l;
    printf("***** performing string length *****\n");
    printf("enter only one string \n");
    scanf("%s",string1);
    l = strlen(string1);
    printf("the string length is %d\n\n",l);
    printf("**** performing string concatenation ****\n");
    printf("enter two strings\n");
    scanf("%s%s",string1,string2);
    printf("the concatenated string is %s\n\n",strcat(string1,string2));
```

C String Functions

```
printf("***** performing string compare *****\n");
printf("enter two strings \n");
scanf("%s%s",string1,string2);
if(strcmp(string1,string2) == 0)
printf("strings are equal\n");
else
printf("strings are not equal\n");
printf("*** performing string copy ***\n");
printf("enter the two strings\n");
scanf("%d%d",string1,string2);
printf("the first string is %s and second string is %s\n",string1,string2);
strcpy(string1,string2);
printf("the first string is %s and second string is %s\n",string1,string2);
}
```

PATTERN MATCHING ALGORITHM

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T . The length of P does not exceed the length of T .

Algorithm: (Pattern Matching)

P and T are strings with lengths R and S , and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T .

1. [Initialize.] Set $K := 1$ and $MAX := S - R + 1$
2. Repeat Steps 3 to 5 while $K \leq MAX$
3. Repeat for $L = 1$ to R : [Tests each character of P]
 If $P[L] \neq T[K + L - 1]$, then: Go to Step 5
 [End of inner loop.]
4. [Success.] Set $INDEX = K$, and Exit
5. Set $K := K + 1$
 [End of Step 2 outer loop]
6. [Failure.] Set $INDEX = 0$
7. Exit

C Program to Find the Character Frequency

```
#include <stdio.h>
int main() {
    char str[1000], ch;
    int count = 0;
    printf("Enter a string: ");
    gets(str);
    printf("Enter a character to find its frequency: ");
    scanf("%c", &ch);
    for (int i = 0; str[i] != '\0'; ++i) {
        if (ch == str[i])
            ++count;
    }
    printf("Frequency of %c = %d", ch, count);
    return 0;
}
```

Output:

Enter a string: This website is awesome.

Enter a character to find its frequency: e

Frequency of e = 4

C Program to Remove all Characters in a String Except Alphabets

```
#include <stdio.h>
int main() {
```

```
    char line[100];
```

```
    printf("Enter a string: ");
```

```
    gets(line);
```

```
    for (int i = 0, j; line[i] != '\0'; ++i) {
```

```
        while (!(line[i] >= 'a' && line[i] <= 'z') && !(line[i] >= 'A' && line[i] <= 'Z') && !(line[i] == '\0')) {
```

```
            for (j = i; line[j] != '\0'; ++j) {
```

```
                line[j] = line[j + 1];
```

```
            }
```

```
            line[j] = '\0';
```

```
        }
```

```
    }
```

```
    printf("Output String: ");
```

```
    puts(line);
```

```
    return 0;
```

```
}
```

Output:

Enter a string: A;'TM...()()E

Output String: ATME

C Program to Copy String Without Using strcpy()

```
#include <stdio.h>
int main()
{
    College of Engineering
```

```
    char s1[100], s2[100], i;
    printf("Enter string s1: ");
    gets(s1);
    for (i = 0; s1[i] != '\0'; ++i) {
        s2[i] = s1[i];
    }

    s2[i] = '\0';
    printf("String s2: %s", s2);
    return 0;
}
```

Output:

Enter string s1: Hey fellow programmer.
String s2: Hey fellow programmer.

```
#include <stdio.h>
int main()
{
```

```
    char line[150];
    int vowels, consonant, digit, space;
    vowels = consonant = digit = space = 0;
    printf("Enter a line of string: ");
    gets(line);
    for (int i = 0; line[i] != '\0'; ++i)
    {
        line[i] = tolower(line[i]);
        if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' || line[i] == 'o' || line[i] == 'u')
        {
            ++vowels;  }
    }
```

```
else if ((line[i] >= 'a' && line[i] <= 'z')) {  
    ++consonant;  
    ATME
```

College of Engineering

```
else if (line[i] >= '0' && line[i] <= '9') {
```

```
    ++digit;
```

```
}
```

```
    else if (line[i] == ' ') {
```

```
        ++space;
```

```
    }
```

```
}
```

```
printf("Vowels: %d", vowels);
```

```
printf("\nConsonants: %d", consonant);
```

```
printf("\nDigits: %d", digit);
```

```
printf("\nWhite spaces: %d", space);
```

```
return 0;
```

```
}
```

Output:

Enter a line of string: ATME college 99

Vowels: 5

Consonants: 6

Digits: 2

White spaces: 2

What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.

Stack has one end, whereas the Queue has two ends (front and rear).

It contains only one pointer top pointer pointing to the topmost element of the stack.

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.

In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

It is called as stack because it behaves like a real-world stack, piles of books, etc.

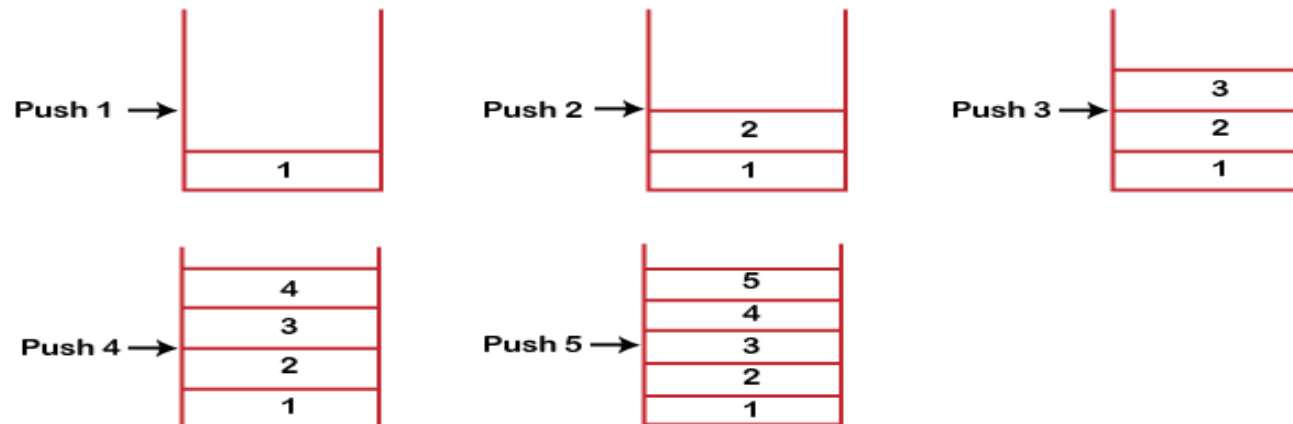
A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

isEmpty(): It determines whether the stack is empty or not.

isFull(): It determines whether the stack is full or not.'

peek(): It returns the element at the given position.

count(): It returns the total number of elements available in a stack.

change(): It changes the element at the given position.

display(): It prints all the elements available in the stack.

Stacks

PUSH operation

The steps involved in the PUSH operation is given below:

Before inserting an element

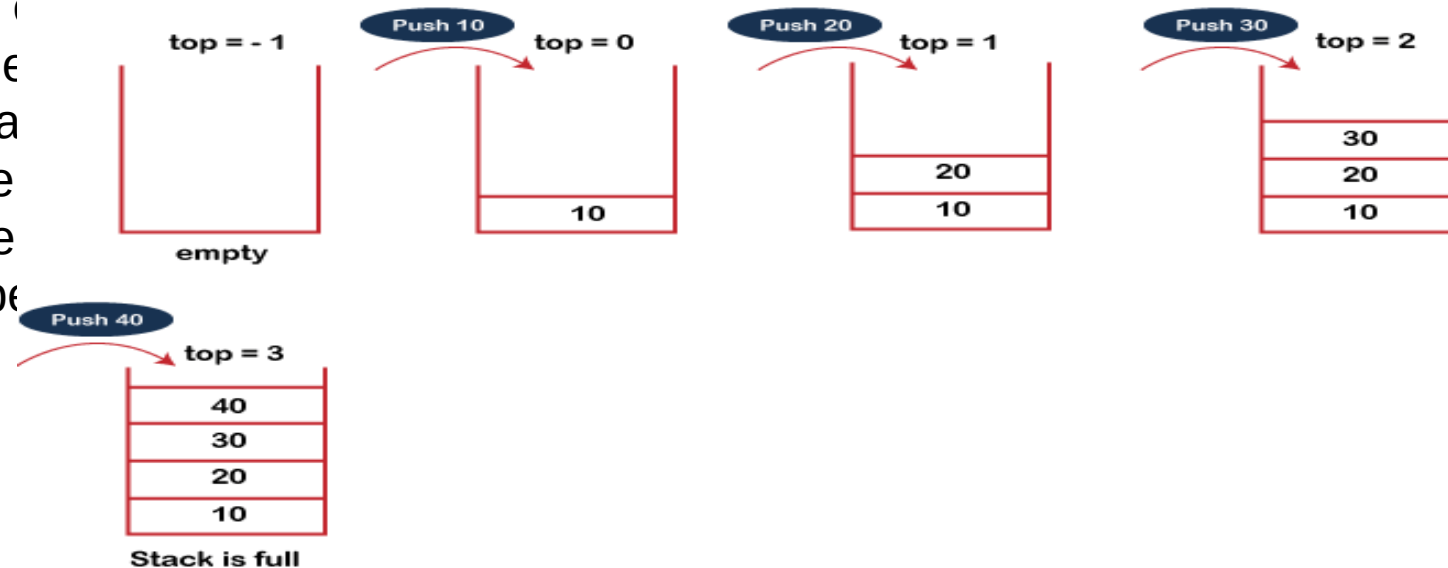
If we try to insert the element

When we initialize a stack

When the new element is added

$top = top + 1$, and the value of top is updated

The elements will be stored in the stack



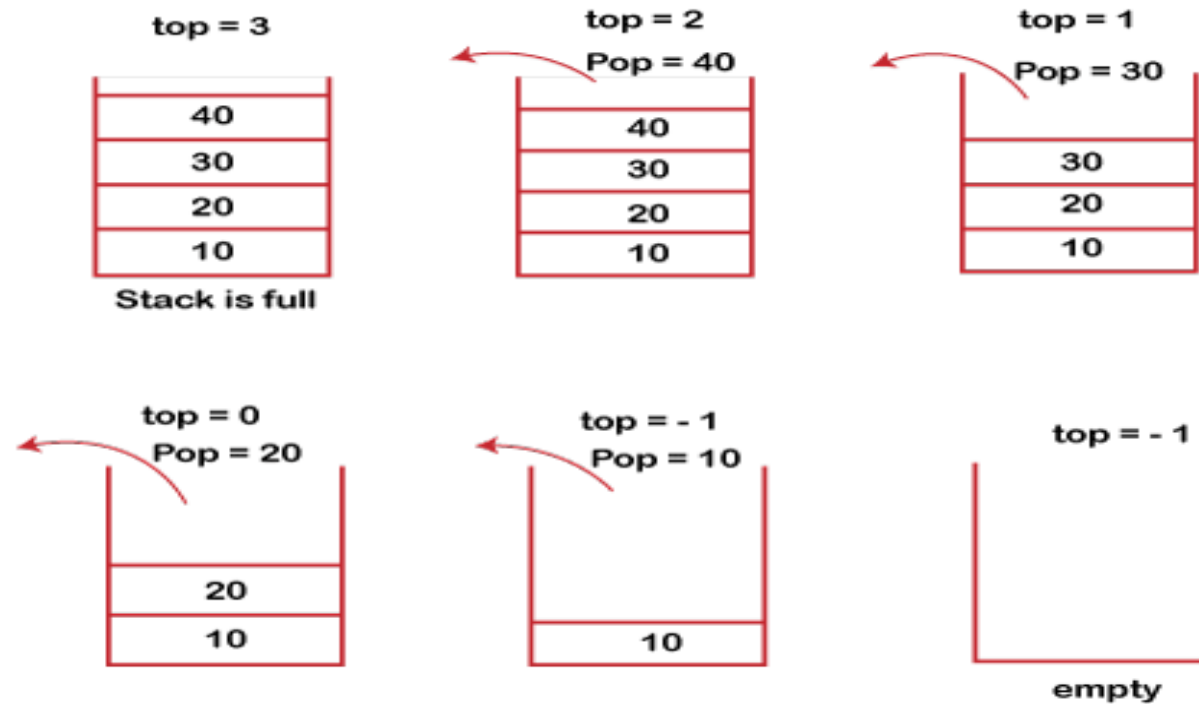
occurs.

mented, i.e.,

Stacks

POP operation

The steps involved
Before deleting the
If we try to delete the
If the stack is not empty
Once the pop operation



1 occurs.

L.

Applications of Stack

Balancing of symbols: Stack is used for balancing a symbol.

String reversal: Stack is also used for reversing a string.

UNDO/REDO: It can also be used for performing UNDO/REDO operations.

Recursion: The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in **which all the previous records of the function are maintained.**

DFS(Depth First Search): This search is implemented on a Graph, and Graph uses the stack data structure.

Backtracking: Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Applications of Stack

Expression conversion: Stack can also be used for expression conversion. This is one of the most important applications of stack.

*Infix to prefix
Infix to postfix
Prefix to infix
Prefix to postfix
Postfix to infix*

Memory management: The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays.

Adding an element onto the stack (**push operation**)

Adding an element into the top of the stack is referred to as push operation. **Push operation** involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
    if (top == n )
        printf("\n Overflow");
    else
    {
        top = top +1;
        stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.

The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The **underflow** condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
  if top = 0 then stack empty;
  item := stack(top);
  top = top - 1;
end;
```

Implementation of POP algorithm using C language

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Stacks

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it.

Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
Begin
  if top = -1 then stack empty
  item = stack[top]
  return item
End
```

Implementation of Peek algorithm in C language

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

Stacks

STACKS USING DYNAMIC ARRAYS

The array is used to implement stack, but the bound (MAX_STACK_SIZE) should be known during compile time.

The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

Stack Operations using dynamic array

1. Stack CreateS()::=

```
typedef struct
{
    int key;        /* other fields */
} element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity= 1;
int top= -1;
```

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= capacity-1;

4. push()

Here the MAX_STACK_SIZE is replaced with **capacity**

```
void push(element item)
{
    /* add an item to the global stack */
    if (top >= capacity-1)
        stackFull();
    stack[++top] = item;
}
```

6. stackFull()

The new code shown below, attempts to increase the **capacity** of the array **stack** so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be.

In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
    REALLOC (stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

$$X = a / b - c + d * e - a * c$$

In above expression contains operators (+, −, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- ☐ Prefix Expression or Polish notation
- ☐ Infix Expression
- ☐ Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

Example: **A + B**

Here, A & B are operands and + is operand

STACK APPLICATIONS: POLISH NOTATION

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: + A B

Here, A & B are operands and + is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: A B +

Here, A & B are operands and + is operand

STACK APPLICATIONS: POLISH NOTATION

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example:

Infix expression: $a/b - c + d * e - a * c$

Fully parenthesized: $((((a/b) - c) + (d * e)) - a * c)$

: a b / e - d e * + a c *

STACK APPLICATIONS: POLISH NOTATION

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.

The expression **$a*(b+c)*d$** which results **$abc+*d*$** in postfix.

Token\	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

STACK APPLICATIONS: POLISH NOTATION

Evaluation of postfix expression using stack

Scan the expression from left to right.

If we encounter any operand in the expression, then we push the operand in the stack.

When we encounter any operator in the expression, then we pop the corresponding operands from the stack.

When we finish with the scanning of the expression, the final value remains in the stack.

Algorithm to evaluate postfix expression

1. Read a character
2. If the character is a digit, convert the character into int and push the integer into the stack.
3. If the character is an operator,
 - Pop the elements from the stack twice obtaining two operands.
 - Perform the operation
 - Push the result into the stack.