Module-1

Define Data:

Data item refers to single unit of values. Considering any employee, hid ID, Name, Phone No, years_of__experience, designation are all examples of data.

Types of Data:

Data can be classified as grouped or elementary data based on the possibility of subdivision. Grouped data can be further subdivided into components. Eg. Name of any employee can be subdivided into First_Name, Middle_Name and Last_Name. However, elementary data cannot be further subdivided. SSN or ID of any employee cannot be further subdivided and is an example for Elementary data.

Each elementary data or component of a grouped data will have a data type.

Problem:
Identifiy the type of data for following:
(i)Aadhar_Card_Number (ii)Name (iii) Age (iv) Weight (v) Address

Solution:

(i) Aadhar_Card_Number - Elementary and Data Type: Long
(ii) Name: Grouped Data with components First_Name, Middle_Name and Last_Name and data type is character array.
(iii) Age: Elementary Data and Data Type: Integer
(iv) Weight: Elementary Data and Data Type: Float
(v) Address: Grouped Data with components Street, City, State, Country, Zip_Code and data type is character array.

Define Entity, Attributes and Entity Set:

Entity is defined as an object of physical or conceptual existence. Eg. Person is an entity of physicial existence and Department is an entity of conceptual existence

Attributes – properties that further describe an entitiy. Eg. Person entity has attributes ID, Name, Phone_No, Email_ID etc.

Entity Set- set of instances of an entity providing values to the attributes of an entity.

Eg:

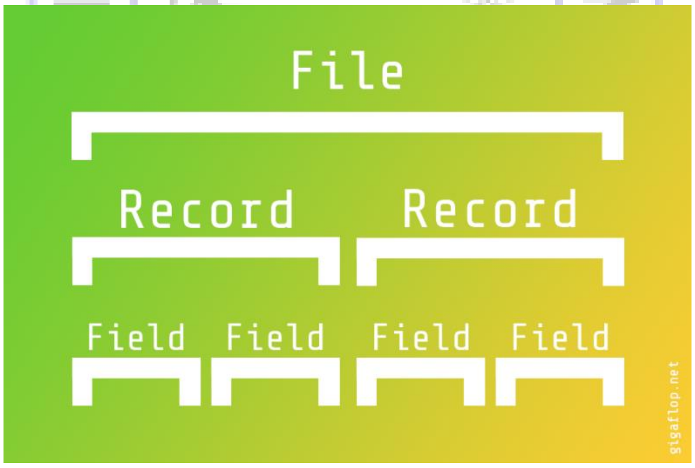
ID	Name	Age	Weight	Address
001	Eena	23	66.2	Hasmukh Nagar, Shimoga, Karnataka, 577202
002	Meena	25	55.1	Ajeeb Nagar, Shimoga, Karnataka, 577203
003	Dika	19	100	Daamu Nagar, Shimoga, Karnataka, 577204

Data vs Information:

Parameter	Data	Information
Definition	Raw facts	Data with context
Context required	Not required as any value starts as data	Always required
Operations	No operations required	Any processing operation like analysis, organization or summarization on data yields information
Relationship between each other	Every data need not be an information	Every information should be on a data
Examples	9900201045, 35, “AIML”, 52.6 are all data	If 9900201045 is given context as Phone Number, it becomes information. Likewise possible context for 35 is age, “AIML” is branch, 52.6 is weight

Files, Records and fields

The data or information collected always need to be physically stored. Such a physical storage is referred as file and referred by a filename. Each file consists of records, where each record captures one entity set. Fields represent attributes of an entity and each record fills the values fieldwise. Hierarchy of files, records and fields is:



Some fields distinguish each record and uniquely define a record. Such fields are called key-fields. Eg: A Student file can be created which captures records of each student in the college. Each student record will have values for fields like USN, Name, Dept, Sem, and Sex. USN is unique for each student and is considered as keyfield.

Fixed-length vs Variable-length records

Parameter	Fixed-length	Variable-length
Definition	Each record is of the same size	The size of each record varies
Fields	All fields should provide fixed length	One variable length field makes record variable length
Storage	Easy to store and acesss	Delemiters are required to separate each record and

		coding is required for processing.
Efficiency	Fixed length can either waste memory or may some times be insufficient	Highly efficient mechanism for storage
Examples	If fields like USN, Sem and Sex is used for Student file, then each record is fixed-length as USN occupies 10 chars, Sem is int, and Sex is 1charachter	If field Name is used in for Student file, then each Name is variable-length and hence also makes variable-length record file.

Problem:
In Hospital patient file: Name, Admission Date, Social Security Number, Room Number, Bed Number, Doctor, identify primary key and identify each data item as grouped or elementary with their types

Social Security Number is unique for each patient and hence can be a primary key
Name – Grouped data with components First_Name, Middle_Name and Last_Name with type character array
Admission Date – Grouped data with components Day, Month and Year with type character array
Social Security Number – Elementary data with type unsigned long
Room Number – Elementary Data with type Integer
Bed Number – Elementary Data with type Integer
Doctor – Grouped data with components First_Name, Middle_Name and Last_Name with type character array

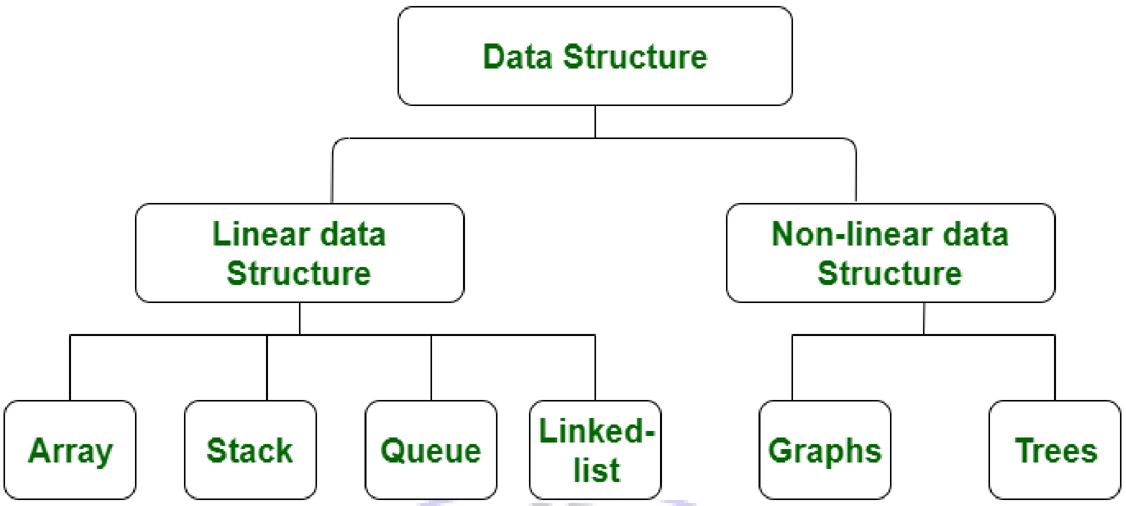
Problem:
Which of following data items may lead to variable-length records when included as items in the record:
(a) Age (b) Sex (c) Name of the spouse (d) Names of children (e) Education (f) Previous employers

Age is fixed size – unsigned integer
Sex – fixed size - one character to store (M, F, and T representing Male, Female and Transgender)
Name of the spouse – variable size
Names of children – variable size
Education – variable size
Previous employers – variable size

Definition of Data Structures
Data structures is defined as a logical or mathematical organization of the data. It can be logical w.r.t numerical index representation in case of arrays or link to next element in case of Linked List. It can be mathematical w.r.t pointer representation in case of trees and graphs.

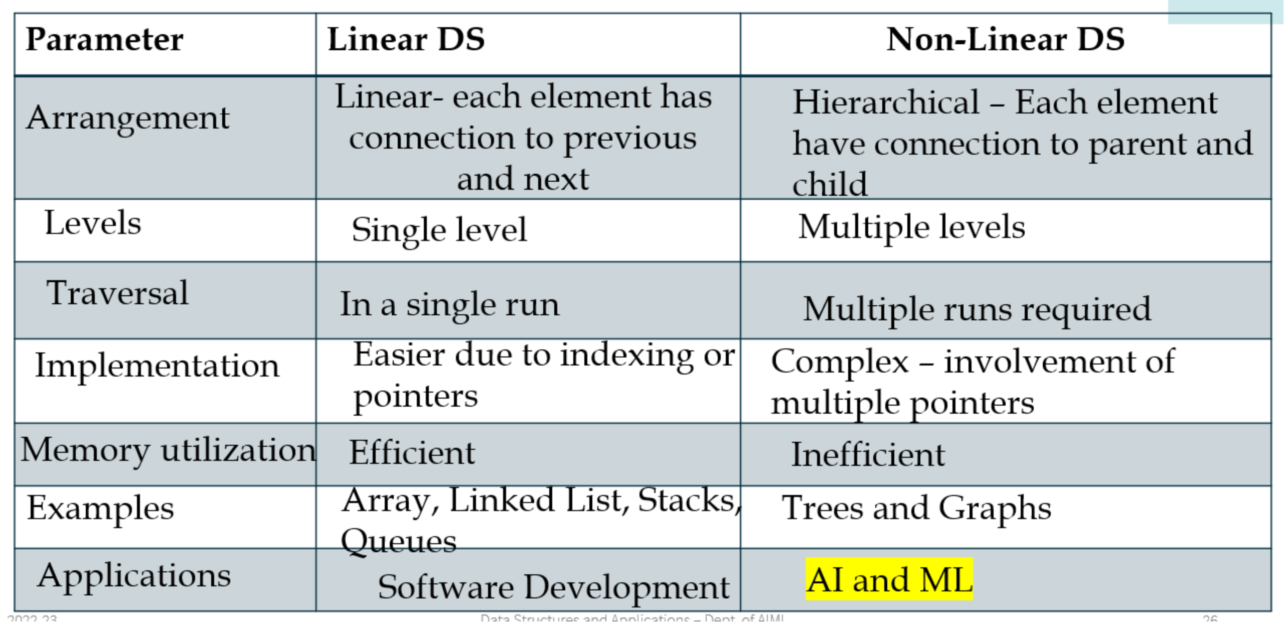
Selection of Data Structures:
Depends on following 2 factors:
(i) Rich enough to represent real world
(ii) Simple enough to process effectively

Classification of Data Structures:



Linear Data Structures have concept of previous and next for any element. In case of arrays, element at any index has previous at (index-1) and next at (index+1). For linked list, there is a link from one element to next element. It is called linear due to contiguous organization in memory. In case of non-linear data structures elements are not connected linearly (contiguously) and exhibit hierarchical relationship.

Differentiate Linear and Non-Linear Data Structures;



Parameter	Linear DS	Non-Linear DS
Arrangement	Linear- each element has connection to previous and next	Hierarchical – Each element have connection to parent and child
Levels	Single level	Multiple levels
Traversal	In a single run	Multiple runs required
Implementation	Easier due to indexing or pointers	Complex – involvement of multiple pointers
Memory utilization	Efficient	Inefficient
Examples	Array, Linked List, Stacks, Queues	Trees and Graphs
Applications	Software Development	AI and ML

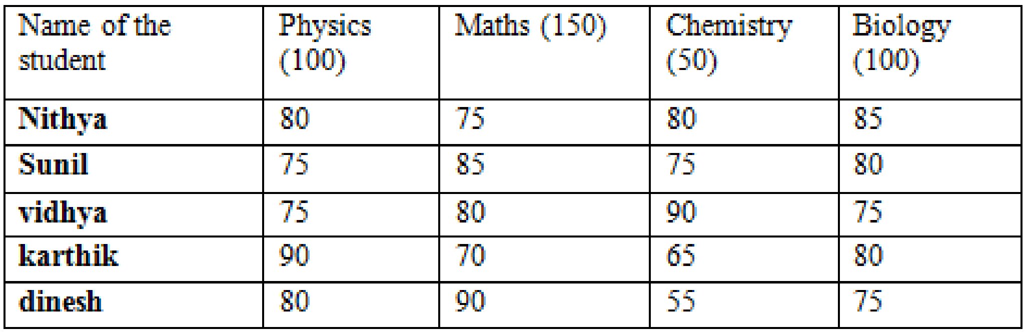
2022-23 Data Structures and Applications – Dept. of AIML 26

Array Data Structures:

Array is a collection of homogenous data elements. Each array has a dimension. Set of numbers can be represented by a 1D array and matrix of numbers can be represented as 2D Arrays. Each array element is accessed through an index that starts with 0. For 1D, one index is used and for 2D arrays, 2 indices are used. Hence number of indices is proportional to number of dimensions of an array.

Real world examples:

Ticket numbers of friends watching movie is an example of 1D array.
Marks of students in various subjects as in:

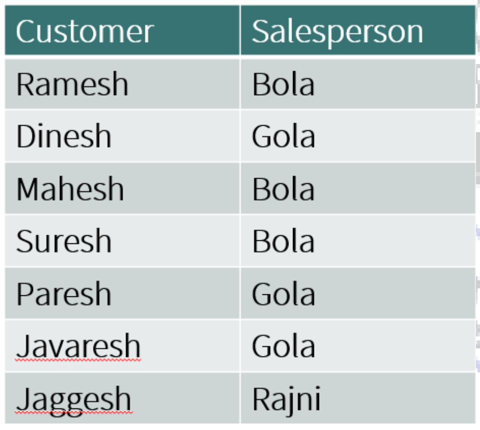


Name of the student	Physics (100)	Maths (150)	Chemistry (50)	Biology (100)
Nithya	80	75	80	85
Sunil	75	85	75	80
vidhya	75	80	90	75
karthik	90	70	65	80
dinesh	80	90	55	75

Here, 2 1D arrays are used:
char names[5][20] – to represent names of students
char subjects[4][20] – to represent names of subjects.
int marks[5][4] – to represent marks of each student (row) subjectwise (col)

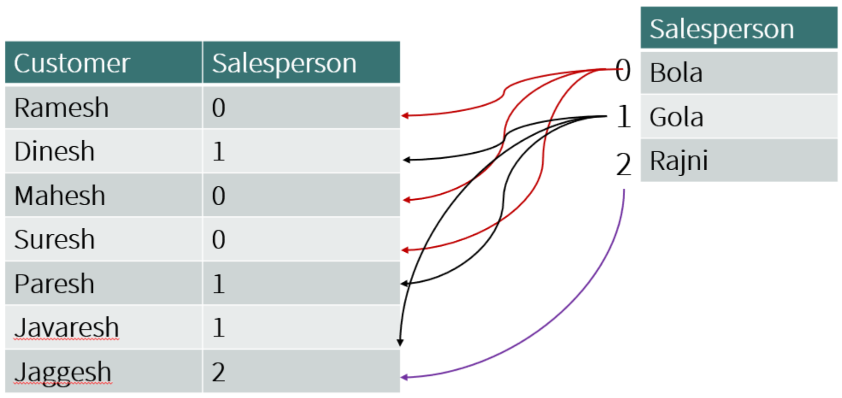
Linked List:
Linked List is a data structure where each element is connected through other element using pointers. If there are set of elements, one element is connected to another in a sequence and last element is connected to NULL.

Realword example:
Consider the following file containing customer and the handling sales person:



Customer	Salesperson
Ramesh	Bola
Dinesh	Gola
Mahesh	Bola
Suresh	Bola
Paresh	Gola
Javaresh	Gola
Jaggesh	Rajni

One way of organizing is use an array for sales person and pointer in the customer file



Customer	Salesperson
Ramesh	0
Dinesh	1
Mahesh	0
Suresh	0
Paresh	1
Javaresh	1
Jaggesh	2

	Salesperson
0	Bola
1	Gola
2	Rajni

Problem with this approach is, if one needs to find all customers of a given sales person, they need to completely traverse the customer file which is highly inefficient when volume of records increases.

Second way of organizing is use an array for sales person and pointer to customers

	Customer
0	Ramesh
1	Dinesh
2	Mahesh
3	Suresh
4	Paresh
5	Javaresh
6	Jaggesh

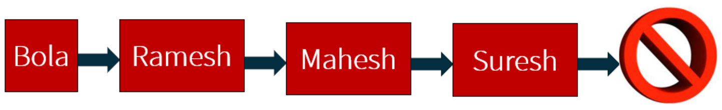
Salesperson	Pointer
Bola	0,2,3
Gola	1,4,5
Rajni	6

Problem with this approach is, that whenever a new customer is added, pointers maintained in salesperson should be changed. Further whenever an existing customer is deleted, pointers need to be removed. The pointer manipulation is a complex operation and hence not efficient.

Hence most efficient way of organization is to use Linked List as shown below:

	Customer	Salesperson
0	Ramesh	2
1	Dinesh	4
2	Mahesh	3
3	Suresh	NULL
4	Paresh	5
5	Javaresh	NULL
6	Jaggesh	NULL

Salesperson	Pointer
Bola	0
Gola	1
Rajni	6



This approach provides advantages like (i) need not traverse entire customer file for searching records and (ii) multiple pointers need not be maintained which is efficient. Hence it overcomes the drawbacks of both previous array based approaches.

Tree data structure:

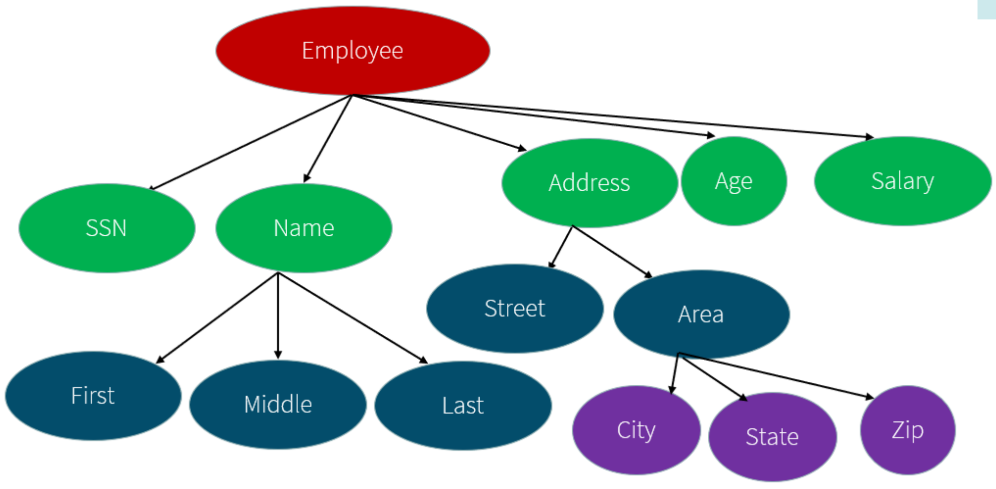
It is a hierarchical data structure representing ancestral relationship (parent, child, grandchild). It has direction from parent to child. The tree logically grows from top to bottom with a master parent called root at the top. Parent of the same children are called siblings. However no direction connection exists between siblings.

Real world examples:

Tree data structure can be applied to capture hierarchical real world scenarios like family tree, organization chart, payment mechanisms, roadmaps and others.

Problem:
Write a tree for:
Employee (SSN, Name, Address, Age, Salary)

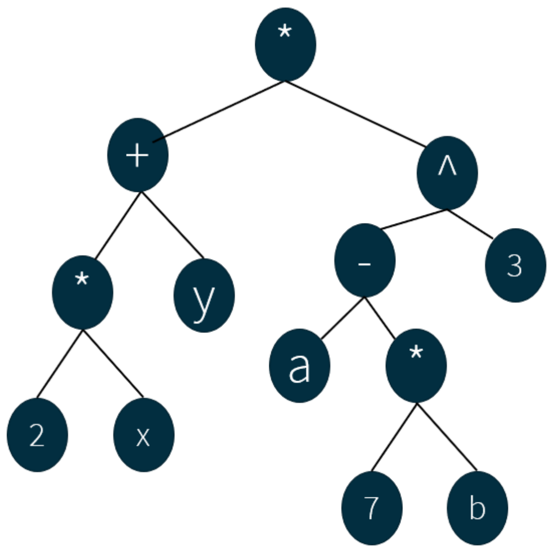
Solution:



This is a 4 level tree. Each subdivision of a field creates a level in the tree.

Problem:
Write a tree for:
(2*x+y)*((a-7*b)^3)

Solution:



While writing tree for expressions, BODMAS rule is used as priority. B-Brackets, O-Order (power), D-Division, M-Multiplication, A-Addition and S-Subtraction.

Stack Data Structures:
A Stack is a special data structure where insertion and deletion happens at one end called TOP. There are many real world scenarios where stack is used like Stack of Disks, Stack of Books, Stack of Plates, Stack of Biscuits and others. Insertion of an element to TOP of the stack is called PUSH and Deletion of an element from the Stack is called POP. It follows LIFO order (Last In First Out).

Queue Data Structure:
Queue is a special data structure where insertion happens at REAR end and deletion happens at FRONT end. There are many real world scenarios where Queue is used like Queue of people for Tickets, Queue of Students near FEE Counter, Queue of People for Lunch, Queue of jobs for printing documents and others. Insertion operation is called ENQUEUE and deletion operation is called DEQUEUE.

Graph Data Structure:
Graph is a special data structure where there can be many to many connections between nodes (elements). The connections can be bidirectional or unidirectional. This data structure is used in Social Networking like Facebook, LinkedIn, Instagram to represent relationship between users.

Difference between Graph and Trees:

Parameter	**Trees**	**Graph**
Relationship	Hierarchical – one way from parent to child	One to many and many to one relationship between elements
Direction	From parent to child	Both bidirectional in which case it is undirected or directional where direction will be explicitly shown
Path from Source to Destination	One unique path	Multiple paths possible

Cycles	No cycles can be formed. Tree is hence called Directed Acyclic Graph	Cycles quite possible
Applications	Hierarchical scenarios like organization chart, family tree, tournament ties copy and others	Social Networking applications like Facebook, Instagram, LinkedIn and others between users.

Data Structure Operations:

Following are the prominent data structure operations:

(i) Searching – search for occurrence of an element in a given set of elements. Result is usually finding existence or not of a given element.

(ii) Traversing – an operation in which one starts with first element, then visits second element, and subsequently all the elements in the order.

(iii) Insertion – an operation to add a new element to a given set of elements. This operation increases the number of elements.

(iv) Deletion – an operation to delete an existing element from a given set of elements. This operation decreases the number of elements.

(v) Updation – an operation to update the value of an existing element in a given set of elements. This operation neither increases nor decreases the number of elements.

(vi) Sorting – Operation for arranging set of elements in increasing (ascending) or decreasing (descending) order.

(vii) Merging – An operation for joining the elements of one set to another.

Problem:

Gym center maintains membership file which contains following information (Name, Address, TelephoneNumber, Age, Sex). Demonstrate all possible operations : Searching, Traversing, Insertion, Deletion, Updation, Sorting and Merging

Solution:

Searching – search for Gym members based on Name or Address or Age or combination of many fields.

Traversing – visit all members from beginning to end stored in the membership file one by one.

Insertion – Add a new member to the Gym

Deletion – Remove an existing member from Gym.

Updation – update the profile of a given member in the Gym

Sorting – Order the members in the Gym in the alphabetical order of Names

Merging – In case Gym has multiple centers, merge records of each file maintained into a single file.

Strings:

Strings are defined as sequence (array) of characters. In C, strings data type does not exist, rather it a character array. In C, all strings are null terminated. Strings are constructed from characters in the character set. Character set followed in C is ASCII (American Standard Code for Information Interchange). ASCII character set is made of alphabets (both upper and lower case), digits and special characters(?,: etc).

Points on Strings:

- A finite sequence of zero or more characters is called Strings.

- String of zero length is called empty string and denoted by "";
- Characters are denoted by single quotes ('A') and strings are denoted by double quotes ("AIML")

String Operations:

- Concatenation: An operation in which second string is appended to the end of first string
  Notation: S1//S2
  Eg: "Hello/World"
- Substring: An operation in which existence of a string inside another string is checked.
  Notation: X//Y//Z
  Where Y is called substring.
  If X is empty, Y is called initial substring
  If Z is empty, Y is called terminal substring

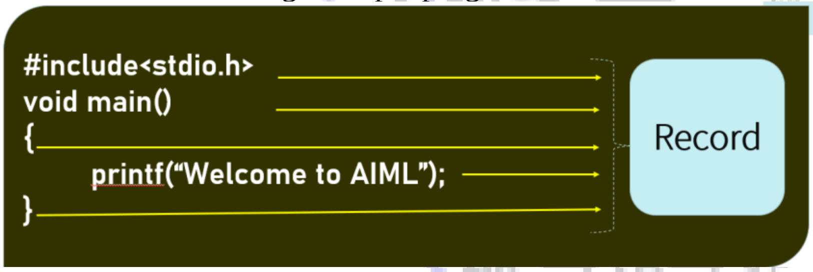
- Eg: In a string "Artificial", "Art" is an initial substring, "cial" is terminal substring and "ifi" is a substring which is neither initial not terminal substring.
- Size of a character in C is 1 byte which is found by calling sizeof(char)

String Storage Options:

Strings are stored in memory in 3 ways:

(i) Record Oriented Fixed Length
(ii) Variable Length storage with fixed maximum
(iii) Linked Storage

Consider the following example program to demonstrate all 3 options:



```
#include<stdio.h>
void main()
{
    printf("Welcome to AIML");
}
```

Each line is called record.

(i) Record oriented Fixed Length

In this storage, each line of program is stored in one record of fixed length. Usually the length is fixed as 80, due to many terminals are of 80 columns. If line length exceeds fixed length, it is stored as multiple records.

Advantages:

- Easy to access any record by providing index value. Further direct access is possible using indexing.
- Ease of updation of any record using index value. This is provided the new record length does not fixed record length.

Drawbacks:

- If a record has lot of blank spaces, lot of time and space wastage occurs.
- In case of records which are lengthy, insertion would require all records to be shifted down.
- Same scenario occurs when records are to be updated and new record length exceeds fixed length.

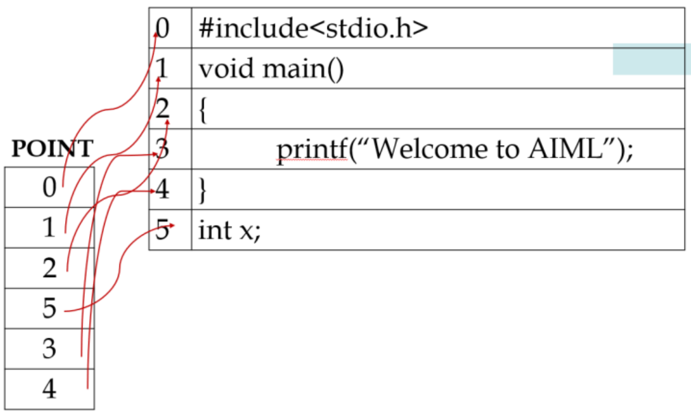
Eg:

Case of Record Insertion

0	#include<stdio.h>
1	void main()
2	{
3	printf(“Welcome to AIML”);
4	}

Inserting a new record in the middle will require bottom records to be shifted by 1

Solution is to maintain a separate POINT array. Whenever new record is added, POINT array can be updated to point to the suitable record.



POINT
0
1
2
5
3
4

0	#include<stdio.h>
1	void main()
2	{
3	printf(“Welcome to AIML”);
4	}
5	int x;

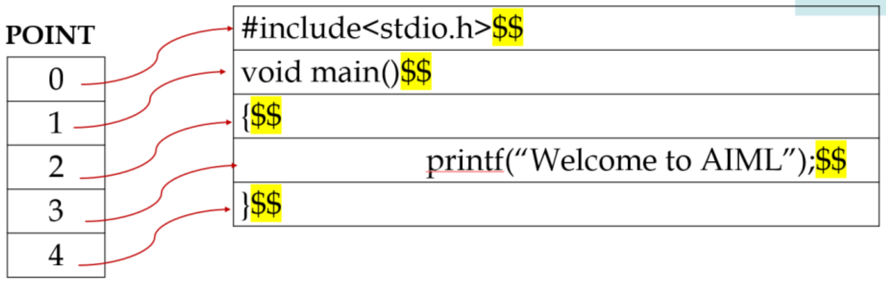
In this example, new record int x; has to be added before printf, This can be adjusted in the POINT array without updating original record lines.

(ii) Variable Length storage with Fixed maximum.

In this approach, each line can be of any length but bounded by fixed maximum. Hence while reading records, only record till its length will be read than fixed length which saves time while accessing it. There are 2 ways of implementing variable length storage:

1. Using markers – Store a marker $$ at end of each record. Markers are also called as sentinels.
2. Using length- Store length separately in POINT array. POINT array has length in first dimension and index in second dimension.

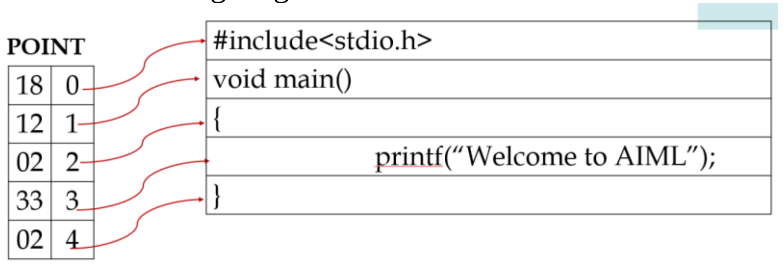
Eg: Using markers



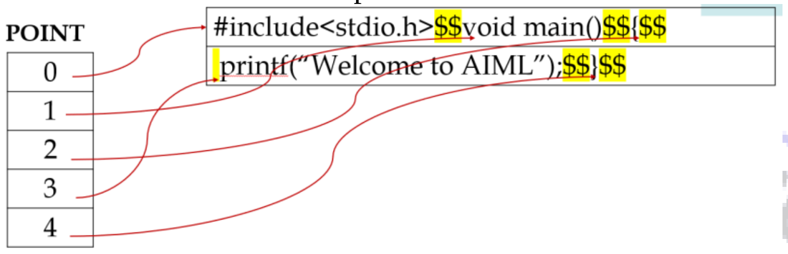
POINT
0
1
2
3
4

#include<stdio.h>$$
void main()$$
{$$
printf(“Welcome to AIML”);$$
}$$

Using lengths



But with this kind of storage, since fixed maximum is used, space is still wasted when record length is very less than fixed maximum. Hence solution is to use variable length storage with markers and variable size. Such an example is shown below:

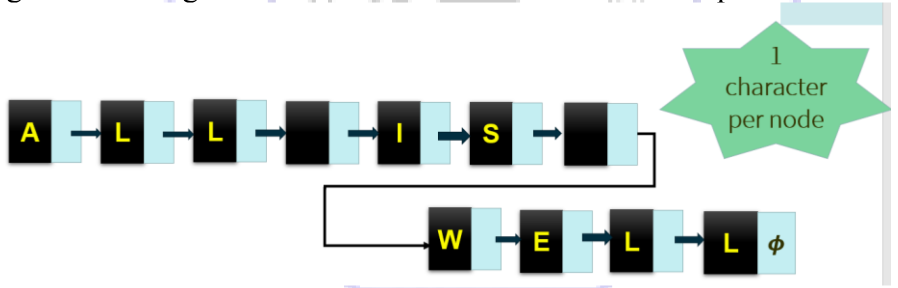

POINT
0
1
2
3
4
#include<stdio.h>$$void main()$${$$
printf("Welcome to AIML");$$}$$

Hence each record can accommodate any number of lines and each occurrence of $$ marks one record.

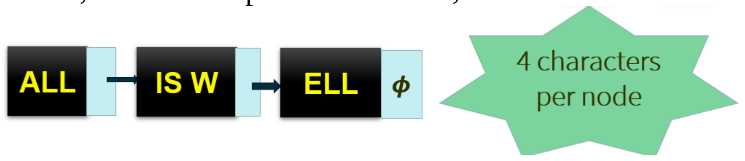
(iii) Linked Storage

In the Linked storage, a linked list is maintained for storage. Linked list consists of node and each node points to a next node. Every node has 'n' number of characters, where 'n' is decided by application.

Eg: Given string is "ALL IS WELL" and for 1 character per node:



In case, 4 character per node is used, then:



The advantage of Linked Storage is it is both time and space efficient.

String operations:

- **SUBSTRING(string, initial, length)**
  Extracts a substring from main string, starting at initial position till number of characters defined by length parameter.
  Eg: SUBSTRING("PANIPURI",3,3) => "NIP"
  SUBSTRING("QUEUE",2,8)=> "UEUE"

- **INDEX(string, pattern)**
  Checks whether pattern is available in the string. If found, returns location of starting character in the pattern. If multiple occurences of same pattern exists, location of first occurrence is returned. If the pattern does not exist, 0 is returned.
  Eg: INDEX("GOBIMANCHURI", "MAN") => Returns 5
  INDEX("QUEUE", "UE") => Returns 2
  INDEX("GOBIMANCHURI", "WOMAN") => Returns 0

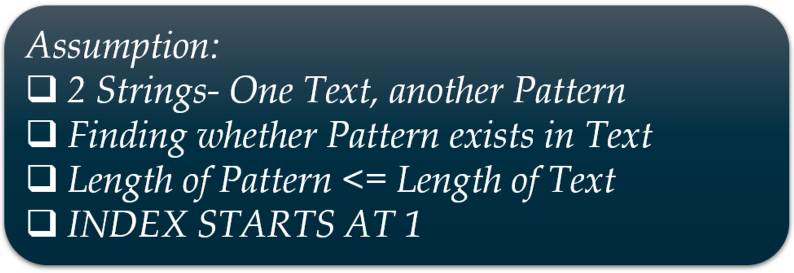
- **CONCAT (S1, S2)**
  String S2 is appended to end of S1.
  Eg: CONCAT("TOM", "JERRY") => TOMJERRY
  CONCAT("JERRY", "TOM") =>JERRYTOM

- **LENGTH(STRING)**
  Returns the number of characters in the String
  Eg: LENGTH("SAMSUNG") returns 7

Pattern Matching Algorithms:

Pattern Matching Algorithms are used check whether a pattern exists in the main string. Such algorithms returns INDEX of occurrence of pattern in main string. In case, pattern is not found, they return 0.

Following assumptions are common for all pattern matching algorithms:



Assumption:
- ❑ *2 Strings- One Text, another Pattern*
- ❑ *Finding whether Pattern exists in Text*
- ❑ *Length of Pattern <= Length of Text*
- ❑ *INDEX STARTS AT 1*

Algorithm of First Pattern Matching scheme:

```
Input: T-Text, P-Pattern, S-LENGTH(T), R-LENGTH(P)
Step 1: Initialize : K=1 and MAX=S-R+1
Step 2: REPEAT STEP 3 to 5 WHILE K<=MAX
Step 3: FOR L=1 to R
            IF P[L]≠ T[K+L-1] THEN GOTO STEP 5
Step 4: SET INDEX=K and EXIT
Step 5: K=K+1
Step 6: INDEX=0
Step 7: EXIT
```

Tracing:

1. T-“JAMBOON”, P-“BOON”

S- LENGTH(T)=7
R- LENGTH(P)=4

Step1: K=1, MAX=S-R+1=7-4+1=4
Step2: WHILE K<=MAX
    1<=4=>TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[1+1-1]=> P[1] ≠ T[1]=>B ≠J => EXIT

K=K+1 => K=1+1=2
2<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[2+1-1]=> P[1] ≠ T[2]=>B ≠A => EXIT

K=K+1 => K=2+1=3
3<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[3+1-1]=> P[1] ≠ T[3]=>B ≠M => EXIT

K=K+1 => K=3+1=4
4<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[4+1-1]=> P[1] ≠ T[4]=>B =B
L=2, P[L] ≠T[K+L-1] => P[2] ≠ T[4+2-1]=> P[2] ≠ T[5]=>O =O
L=3, P[L] ≠T[K+L-1] => P[3] ≠ T[4+3-1]=> P[3] ≠ T[6]=>O =O
L=4, P[L] ≠T[K+L-1] => P[4] ≠ T[4+4-1]=> P[4] ≠ T[7]=>N =N

SET INDEX=K => INDEX=4

K=K+1 => K=4+1=5
5<=4 => FALSE

RETURN INDEX which is 4

2. T-“JAMBOON”, P-“BONE”

S- LENGTH(T)=7
R- LENGTH(P)=4

Step1: K=1, MAX=S-R+1=7-4+1=4
Step2: WHILE K<=MAX
    1<=4=>TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[1+1-1]=> P[1] ≠ T[1]=>B ≠J => EXIT

K=K+1 => K=1+1=2

2<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[2+1-1]=> P[1] ≠ T[2]=>B ≠A => EXIT

K=K+1 => K=2+1=3
3<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[3+1-1]=> P[1] ≠ T[3]=>B ≠M => EXIT

K=K+1 => K=3+1=4
4<=4 => TRUE
FOR L=1 to R=> FOR L=1 to 4
L=1, P[L] ≠T[K+L-1] => P[1] ≠ T[4+1-1]=> P[1] ≠ T[4]=>B =B
L=2, P[L] ≠T[K+L-1] => P[2] ≠ T[4+2-1]=> P[2] ≠ T[5]=>O =O
L=3, P[L] ≠T[K+L-1] => P[3] ≠ T[4+3-1]=> P[3] ≠ T[6]=>N ≠O =>EXIT

K=K+1 => K=4+1=5
5<=4 => FALSE

RETURN INDEX which is 0

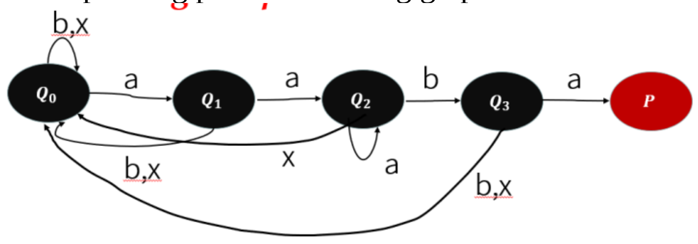
Second Pattern Matching Algorithm:
In first algorithm, pattern is checked in the text and compared iteratively to check pattern exists or not. In second algorithm, text is compared with pattern.

For each pattern, a pattern matching table is created in which columns are unique symbols of table along with an 'x' which represents a symbol not in the pattern. Rows of the table corresponds to the states, where initial state is Qo and final state is P. Number of states is equal to the number of characters in the pattern.

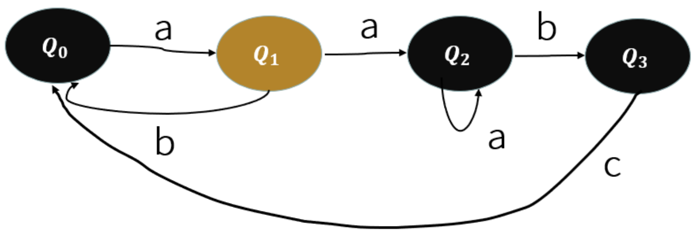
Eg: P=aaba
Construction of Pattern matching table:

State	**a**	**b**	**x**
QO	**Q1**	QO	QO
Q1	**Q2**	QO	QO
Q2	Q2	**Q3**	QO
Q3	**P**	QO	QO

Corresponding pattern matching graph:

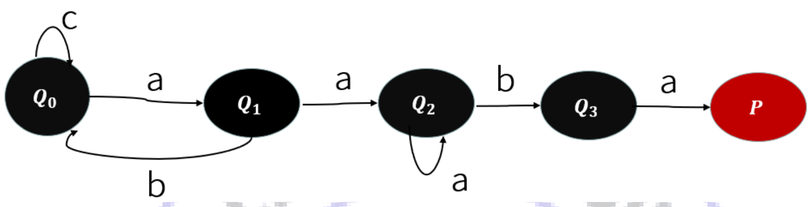


Check for patterns:
(i) T=aabcaba



Since, it lands at an intermediate state, pattern is not found

(ii) T=abcaabaca



Since, it lands at final state P, pattern is found.

Algortihm:

```
Input: T-Text, P-Pattern, S-LENGTH(T), R-LENGTH(P)
Step 1: Initialize : K=1 and S1= $Q_0$
Step 2: REPEAT STEP 3 to 5 WHILE $S_K$ ≠P and K<=N
Step 3: READ $T_K$
Step 4: SET $S_{K+1} = F(S_K, T_K)$
Step 5: SET K=K+1
Step 6: IF $S_K = P$ then
              INDEX=K
         ELSE
              INDEX=0
Step 7: EXIT
```

Linear Arrays:
Linear Arrays are a list of finite number of n homogeneous elements such that:
(a) Elements referred by index of n consecutive numbers
(b) Elements stored in contiguous memory allocation

Eg: int a[4] is a linear array as:
it is finite (4 elements), homogeneous as each element is integer, referred by index set {0,1,2,3} and allocated contiguously in stack. If array a starts with address 1012, next element will be at 1016, then 1020 and last element at 1024.

Number of elements in linear array is computed using formula:

Length=UB-LB+1

where,
UB – Upper Bound is the highest index of the array

LB – Lower Bound is the lowest index of the array (0 in many languages, whereas 1 in other langauges. Can also be negative if language supports negative indexing)

Eg for int a[4], UB=3 and LB=0 Hence Length=UB-LB+1=3-0+1=4

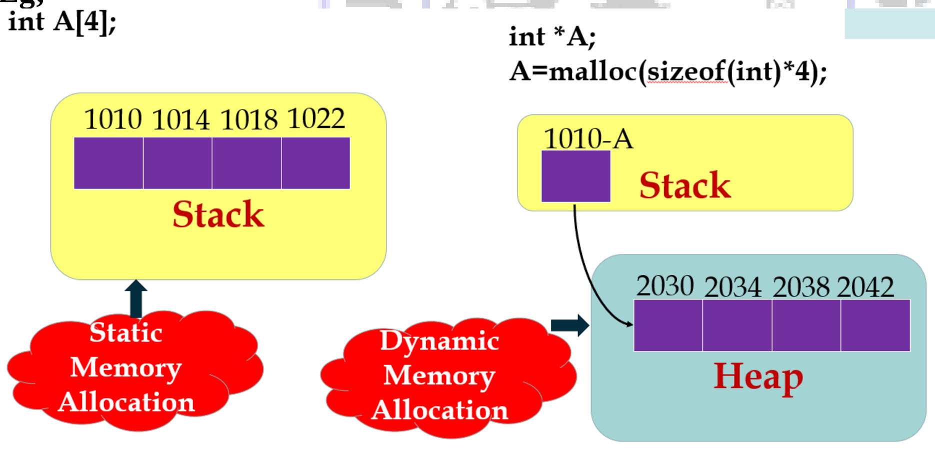
Further, for linear arrays, concept of subscript and subscripted variables exists.
In case of int a[4], one can access third element of an array as a[2].
Here 2 is the subscript and a[2] is called subscripted variable.

Linear Array Declaration:
Each declaration of an array has 3 pieces of information:

1. Name of the array
2. Data type of the array
3. Index set of the array

Eg:
For array declaration float points[5],
Name is points, data type is float and index set is {0,1.2.3.4}
Linear Arrays can be accessed in constant time, means to access any element in the array same time is required. Eg. int a[100], to access first element (a[0]). Fiftieth element (a[49]) and 100th element (a[99]) same time is required.

Representation of Linear Array in Memory:
There are two types of declaration of Linear Arrays: (i) Static and (ii) Dynamic.
In case of static memory allocation, stack is used and in case of dynamic, heap is used.
Eg;



Differences between Static and Dynamic memory allocation of Linear Arrays:

Parameter	Static Memory Allocation	Dynamic Memory Allocation
Memory Area	Stack is used	Stack is used to store pointer to the array and array is stored in heap
Number of elements	Fixed during compile time	Can be changed during runtime
Need of any functions	No, just mention the number of elements while declaring then arrays	Yes, malloc() for memory allocation and free() to clear the same

Garbage formation	No, as the stack is cleared when the function/program exits	If free() is not called, the array in the heap becomes garbage after function/program exits
Space Efficiency	Becomes fixed and can lead to either too much or too less	Consumes as much space required
Overhead	No overheads	Maintaining an additional pointer in the stack to refer to array in the heap

Example	int A[4]	int *A;
		A=malloc(sizeof(int)*4);

Programs/Algorithms on Linear Array Operations:

1. **Creating and Displaying Array:**

```
Algorithm for Creating Array:
Step 1: Start
Step 2: Read the size of the array (N) as input from the user
Step 3: Allocate the required memory (Specified size * Data type size) and store pointer as ARR
Step 4:For I=0 to N-1
        4.1: Read element from keyboard
        4.2: Store the element in ARR[I]
Step 5: Return
```

```
Algorithm for Displaying the Array:
Step 1: Start
Step 2: For I=0 to N-1
        2.1: Print the element in ARR[I]
Step 3: Return
```

```
#include<stdio.h>
void create_array(int *arr,int n)
{
   printf("Enter %d array elements one by one\n",n);
   int i;
   for(i=0;i<n;i++)
   {
      scanf("%d",&arr[i]);
   }
}

void display_array(int *arr,int n)
{
   printf("Array Elements are \n");
   int i;
   for(i=0;i<n;i++)
   {
      printf("%d\n",arr[i]);
   }
}


void main()
{
   int choice,n,sum,min;
   int *arr;
   while(1)
   {
   printf("***********Array Menu Operations*******\n");
   printf("1. Create\n");
   printf("2. Display\n");
   printf("3. Exit\n");
   printf("Enter your choice\n");
   scanf("%d",&choice);

   switch(choice)
   {
       case 1: printf("Create an array\n");
                  printf("Enter the value of n\n");
                  scanf("%d",&n);
                  arr=malloc(sizeof(int)*n);
                  create_array(arr,n);
                    break;
      case 2:printf("Display the array\n");
                display_array(arr,n);
                 break;
      case 3:exit(0);
      default:printf("Enter 1 2 or 3\n");

   }
   }
}
```

2. Bubble Sort

Algorithm:

```
Step 1: Start
Step 2: Initialize an array ARR of ‘N’ elements
Step 3: For I=0 to N-1  //Passes
          Step 3.1: FOR J=0 to N-1-I  //Comparisons
                    Step 3.1.1: IF ARR[J] > ARR[J+1]
                                   SWAP ARR[J] and ARR[J+1]
 Step 4: FOR I=0 to N-1
          Step 4.1 PRINT ARR[I]
 Step 5: Return
```

C Program:

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

```
void main()
{

    int a[5]={1,3,2,4,0};
    int N=5,i,j,t;
    //Loop for passes
    for(i=0;i<N-1;i++)
    {
        //Loop for comparisons
        for(j=0;j<N-i-1;j++)
        {
            //Check for swapping
            if(a[j]>a[j+1])
            {
                //Do swapping
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
        }

    }
    //Print sorted array
    for(i=0;i<N;i++)
    {
        printf("%d\n",a[i]);
    }
}
```

3. Linear Search

Algorithm:

```
Step 1: Start
Step 2: Initialize an array ARR of 'N' elements
Step 3: Read key KEY from the user
Step 4: For I=0 to N-1  //Searching Loop
            Step 4.1: IF ARR[I]==KEY THEN
                          PRINT "SUCCESSFUL SEARCH"
                          RETURN
Step 5: PRINT "UNSUCCESSFUL SEARCH"
Step 6: Return
```

C Program:

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

    int a[5]={1,3,2,4,0};
    int KEY,i,N=5;
    printf("Enter the key\n");
```

```
   scanf("%d",&KEY);
   //Loop for passes
   for(i=0;i<N;i++)
   {
      if(a[i]==KEY)
      {
         printf("Succesful Search\n");
         exit(0);
      }
   }
   printf("Unsuccesful Search\n");
}
```

Multidimensional Arrays:

Multidimensional Arrays are the linear arrays with number of dimensions more than 1. Few examples are matrix of 2 dimension (with rows and columns), image of 3 dimension (channels, rows, columns), video of 4 dimension (frame, channels, rows, columns).

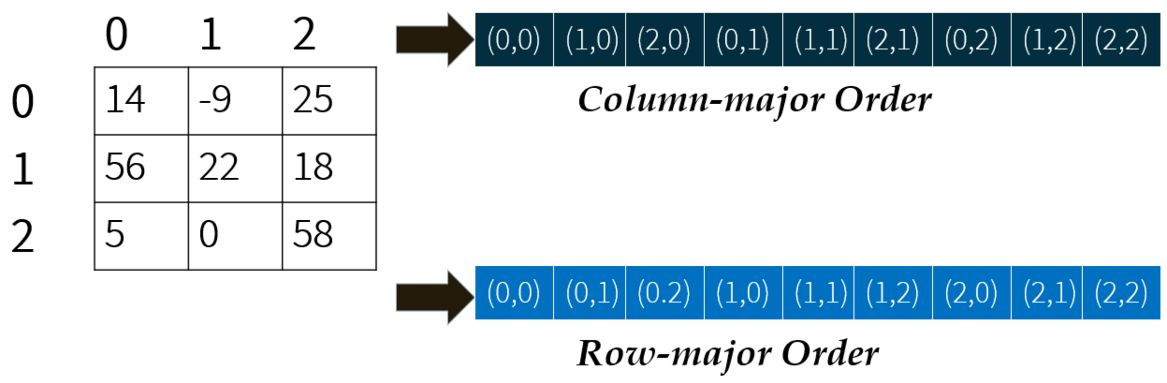
Representation of 2D Arrays in memory:

There are 2 representations: Row-major order and Column-major order.

In the row-major order, the elements are stored row by row. First row elements are filled, subsequently second row elements till last row.

In the column-major order, the elements are stored column by column. First column elements are filled, subsequently second column elements till last column.

Eg:



Programs/Algorithms on 2D Arrays:

1. **Creating 2D array statically, reading input from keyboard, displaying matrix and computing trace**

Algorithm:

```
Step 1: Start
Step 2: Create 2D array MAT with size M and N
Step 3: FOR I=0 to M-1
        Step 3.1: FOR J=0 to N-1
                Step 3.1.1: Read element from keyboard and store into MAT[I][J]
Step 4: FOR I=0 to M-1
        Step 4.1: FOR J=0 to N-1
```

```
                Step 4.1.1: PRINT MAT[I][J]
Step 5: FOR I=0 to M-1
        Step 5.1: SUM=SUM+ MAT[I][I]
Step 6: PRINT SUM
Step 7: Return
```

C Program:

```
#include<stdio.h>
void main()
{
    int M=3,N=3;
    int mat[M][N],i,j;

    int sum=0;
    printf("Enter the matrix elements\n");
    for(i=0;i<M;i++)
    {
       for(j=0;j<N;j++)
       {
          scanf("%d",&mat[i][j]);

       }
    }
    printf("Matrix:\n");
    for(i=0;i<M;i++)
    {
       for(j=0;j<N;j++)
       {
          printf("%d ",mat[i][j]);

       }
       printf("\n");
    }
    for(i=0;i<M;i++)
    {
       sum=sum+mat[i][i];
    }
    printf("Trace of matrix=%d\n",sum);

}
```

2. Creating 2D array dynamically, reading input from keyboard, displaying matrix and computing trace

Algorithm:

```
Step 1: Start
Step 2: Get M (no of rows) and N (no of cols) from user
Step 3: Create a pointer to point to a pointer array MAT
Step 4: FOR I=0 to M-1
        Step 4.1: Allocate memory for each element of MAT[i]
Step 5: FOR I=0 to M-1
          Step 3.1: FOR J=0 to N-1
                    Step 3.1.1: Read element from keyboard and store into MAT[I][J]
Step 6: FOR I=0 to M-1
          Step 4.1: FOR J=0 to N-1
                    Step 4.1.1: PRINT MAT[I][J]
Step 7: FOR I=0 to M-1
          Step 5.1: SUM=SUM+ MAT[I][I]
 Step 8: PRINT SUM
 Step 9: Return
```

C Program:

```
#include<stdio.h>
void main()
{
    int **mat,i,j,M,N;
    printf("Enter the value of M and N\n");
    scanf("%d%d",&M,&N);
    mat=malloc(sizeof(int*)*M);
    for(i=0;i<M;i++)
    {
       mat[i]=malloc(sizeof(int)*N);
    }
    int sum=0;
    printf("Enter the matrix elements\n");
    for(i=0;i<M;i++)
    {
       for(j=0;j<N;j++)
       {
          scanf("%d",&mat[i][j]);

       }
    }
    printf("Matrix:\n");
    for(i=0;i<M;i++)
    {
       for(j=0;j<N;j++)
       {
          printf("%d ",mat[i][j]);

       }
       printf("\n");
    }
    for(i=0;i<N;i++)
    {
       sum=sum+mat[i][i];
    }
    printf("Trace of matrix=%d\n",sum);
}
```

3. C Program to compute trace of a dynamically allocated 2D Array

```
Algorithm:
Step 1: Start
Step 2: Set M to no of rows and N to no of cols
Step 3: Create a pointer to point to a pointer array MAT
Step 4: FOR I=0 to M-1
        Step 4.1: Allocate memory for each element of MAT[i]

Step 5: FOR I=0 to M-1
        Step 3.1: FOR J=0 to N-1
                Step 3.1.1: Read element from keyboard and store into MAT[I][J]
Step 6: FOR I=0 to N-1  //Iterate till no of columns
        Step 4.1: FOR J=0 to M-1   //Iterate till no of rows
                Step 4.1.1: PRINT MAT[J][I]    //Change of row and column index
Step 7: Return
```

```
#include<stdio.h>
void main()
{
   int **mat;
   int i,j,m=3,n=2;
   mat=malloc(sizeof(int*)*m);
   for(i=0;i<m;i++)
   {
      mat[i]=malloc(sizeof(int)*n);
   }
   printf("Enter the matrix values one by one\n");
   for(i=0;i<m;i++)
   {
      for(j=0;j<n;j++)
      {
         scanf("%d",&mat[i][j]);
      }
   }
   printf("Transpose of matrix\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         printf("%d ",mat[j][i]);
      }
      printf("\n");
   }
}
```

Address computation of an array element:

Address of any element of an array can be computed using formulae:

$$Address\ of\ A[k] = BA(A) + w(k - LB)$$

where,
k – subscript
BA – Base Address
w – word length- 2bytes for int, 4bytes for float (32 bit compilers)
LB – lower bound

Eg. Consider array declaration: int A[10] and base address=1012
Address of A[7]=1012+2*(7-0) = 1012 + 14 =1026

Storing the values of an array:

There are three techniques to store array values:
(i) Declaration with initialization:
Eg: int a[4]={1,2,3,4} OR int a[]={1,2,3,4}. Both syntax are correct. In first case, programmer specifies the size and in latter case compiler automatically computes the size.

(ii) Read from keyboard:
Loops are used with indexing to read all the values of an array from keyboard.

```
Eg: for(i=0;i<n;i++)
          scanf("%d",&a[i]);
```

(iii) Assign individually
The subscript of the array can be used to store each element individually.

```
Eg: int a[4];
      a[0]=10;
      a[1]=20;
      a[2]=14;
      a[3]=22;
```

Write a C program to demonstrate full array copying:

```
#include<stdio.h>

void main()
{
   int i;

  int a[]={1,2,3,4,5,6,7,8};
  int b[8];
   for(i=0;i<8;i++)
   {
      b[i]=a[i];  //Copying code
   }
   for(i=0;i<8;i++)
        printf("%d\n",b[i]);

}
```

Write a C program to demonstrate selective copying of an array

```
#include<stdio.h>

void main()
{
    int i;

   int a[]={1,2,3,4,5,6,7,8};
   int b[4],j=0;
    for(i=0;i<8;i++)
    {
        if(a[i]%2==0) //Only copy Even number
        {
            b[j++]=a[i];
        }
    }
    for(i=0;i<4;i++)
         printf("%d\n",b[i]);

}
```

Jagged Arrays:

A double pointer can be created in stack which can point to Pointer Arrays. Each pointer in the pointer array can point to arrays of variable size. An array in which each row contains variable number of columns is called “Jagged Arrays”.

C Program to demonstrate jagged arrays:

```
#include<stdio.h>
void main()
{
    char **jag; //Jagged Array
    int i,j;
    jag=malloc(sizeof(char*)*4);  //Allocate memory for rows
    for(i=0;i<4;i++)
    {
        jag[i]=malloc(sizeof(char)*(i+1)); //Allocate memory for columns of each row

    }
    for(i=0;i<4;i++)
    {
       for(j=0;j<=i;j++)  // j depends on i
       {
           jag[i][j]='*';
       }

    }
    for(i=0;i<4;i++)
    {
       for(j=0;j<=i;j++)
       {
           printf("%c ",jag[i][j]);
       }
       printf("\n");
    }
}
```

C Program to create following patterns using jagged arrays:

```
*
*  *
*  *  *
*  *  *   *
```

```
#include<stdio.h>
void main()
{
   char **jag; //Jagged Array
   int i,j;
   jag=malloc(sizeof(char*)*4);  //Allocate memory for rows
   for(i=0;i<4;i++)
   {
      jag[i]=malloc(sizeof(char)*(i+1)); //Allocate memory for columns of each row

   }
   for(i=0;i<4;i++)
   {
     for(j=0;j<=i;j++)  // j depends on i
     {
        jag[i][j]='*';
     }
   }
   for(i=0;i<4;i++)
   {
     for(j=0;j<=i;j++)
     {
        printf("%c ",jag[i][j]);
     }
     printf("\n");
  }
}
```

Structures:

Structures are collection of heterogeneous elements. Arrays are collection of homogeneous elements. Heterogeneity of structures is with respect to different type of data packed in a structure.

Eg:

```
struct person
{
    char name[10];
    int age;
    float salary;
};
```

Here person is called tag of structure and name, age and salary are called fields.

Hence a structure in this example holds different types of data like character array, integer and floating point.

To create a variable of structure type, syntax is:
struct person p1;

To assign the values to fields of structure use "." operator:

```
strcpy(p1.name,"manu")  //Note strings cant be directly assigned, You have to use strcpy
p1.age=22;
p1.salary=25000.0;
```

To create an alias for a structure, typedef is used.
Eg: instead of struct person one can alias it to person with following syntax:

```
typedef struct
{
    char name[10];
    int age;
    float salary;
}person;
```

Note: while computing the size of structure, for a character array take size in multiple of word boundary (multiple of 4)

C program to check whether 2 persons are identical:

```
#include<stdio.h>

typedef struct
{
   char name[20];
   int age;
   float salary;

}person;

void comparePersons(person p1, person p2)
{
   if(strcmp(p1.name,p2.name)==0  &&  p1.age==p2.age &&  p1.salary==p2.salary)
   {
      printf("Both are same\n");
   }
   else
   {
      printf("Both are different");
   }
}
void main()
```

```
{
   person p1={"ramu",22,20000};
   person p2={"ramu",22,20000};
   comparePersons(p1,p2);
}
```

Write a C program to create an array of 3 persons, read the data and print the same – Use static allocation

```
#include<stdio.h>

typedef struct
{
   char name[20];
   int age;
   float salary;

}person;

void main()
{
   int i;
   person p[3]; //Static allocation by Declaration
  //Read the data
   printf("Enter details of 3 person\n");
   for(i=0;i<3;i++)
   {
      scanf("%s%d%f",p[i].name,&p[i].age,&p[i].salary);
   }
   //Print the data
   for(i=0;i<3;i++)
   {
     printf("%s,%d,%f\n",p[i].name,p[i].age,p[i].salary);
   }
}
```

Write a C program to create an array of n persons, read the data and print the same – Use dynamic allocation.

```
#include<stdio.h>
typedef struct
{
   char name[20];
   int age;
   float salary;
}person;


void main()
{
   int i,n;
   printf("Enter the value of n \n");
   scanf("%d",&n);
   person *p; //Declaration
   p=malloc(sizeof(person)*n);
   //Read the data
   printf("Enter details of %d person\n",n);
   for(i=0;i<n;i++)
   {
       scanf("%s%d%f",p[i].name,&p[i].age,&p[i].salary);
   }
   //Print the data
   for(i=0;i<n;i++)
   {
      printf("%s,%d,%f\n",p[i].name,p[i].age,p[i].salary);
   }
 free(p);//Ensures no garbages are created
}
```

Nested Structures:

One can put a structure inside another and this concept is called Nested structures.

Previous program can be modified with person structure containing a structure for date of birth as follows:

```
#include<stdio.h>
typedef struct
{
    int day;
    int month;
    int year;
}dob;
typedef struct
{
    char name[20];
    int age;
    float salary;
    dob d; //Since aliased dob can be directly used instead of struct dob
}person;  //Person is a nested structure containing DOB

void main()
{
    int i,n;
    printf("Enter the value of n \n");
    scanf("%d",&n);
    person *p; //Declaration
    p=malloc(sizeof(person)*n);
    //Read the data
    printf("Enter details of %d person\n",n);
    for(i=0;i<n;i++)
    {
       scanf("%s%d%f%d%d%d",p[i].name,&p[i].age,&p[i].salary,
            &p[i].d.day,&p[i].d.month,&p[i].d.year);
    }
    //Print the data
    for(i=0;i<n;i++)
    {
      printf("%s,%d,%f,%d,%d,%d\n",p[i].name,p[i].age,p[i].salary ,
            p[i].d.day,p[i].d.month,p[i].d.year);
    }

  free(p);//Ensures no garbages are created
}
```

Structures v/s Unions

Sl. No.	Parameter	Structure	Union
3.	Keyword	struct keyword is used to define structures	union keyword is used to define union
4.	Memory size	The total size of a structure is equal to the sum of the different fields in a structure	The total size of a union is equal to the maximum of the different fields in a union
5.	Memory Allocation	Separate block of memory is allotted to each field	Single block of memory is allocated which is shared by all the fields

6.	Example size computation	typedef struct
		{
		  char name[22];
		  int age;
		  int rank;
		}person;
		Size=24(due to multiple of 4 rule)+4+4=32
			typedef union
			{
			  char name[22];
			  int age;
			  int rank;
			}uperson;
			Size=24(due to multiple of 4 rule)

7.	Access	Simultaneous access of multiple members is possible	Only one member can be accessed at each time. Other fields become garbages at that time
8.	Change of value	Multiple members value can be changed.	Only one member value can be changed at any time and other members then become garbage.

Self Referential Structures:

Structures in which a field refers the defined structure is called Self Referrential structures. Eg:

```
typedef struct
{
    int data;
    struct list *next;
} list;

void main()
{
    list *l1,*l2,*l3;
    l1=malloc(sizeof(list));
    l2=malloc(sizeof(list));
    l3=malloc(sizeof(list));
    l1->data=20;
    l1->next=l2;
    l2->data=40;
    l2->next=l3;
    l3->data=10;
    l3->next=NULL;

}
```

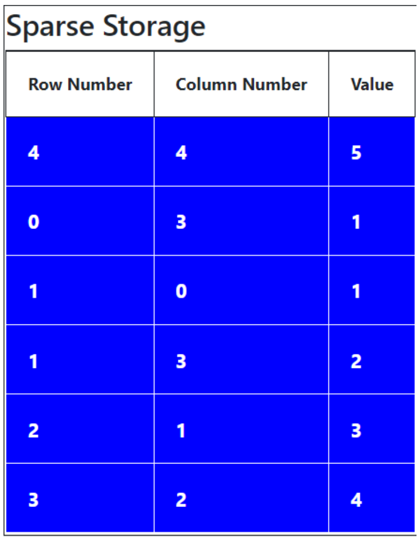
Here structure list contains a pointer next which refers to structure list itself and hence the structure is called Self Referential structure. Further this structure represents one node and multiple nodes (l1, l2,l3) is created. Dynamically memory is allocated for each node. The pointers create a link between nodes and resulting structure is called linked list.

Sparse Matrix Representation using Arrays

Sparse Matrix is a special matrix which consists of most of zero elements and few non-zero elements. Hence instead of 2D representation for sparse matrix, a structure based representation can be used to efficiently represent the matrix, The structure consists of fields namely row, column and value,

Eg. Consider a sparse matrix of size 4X4
0 0 0 1
1 0 0 2
0 3 0 0
0 0 4 0

The corresponding sparse storage is:

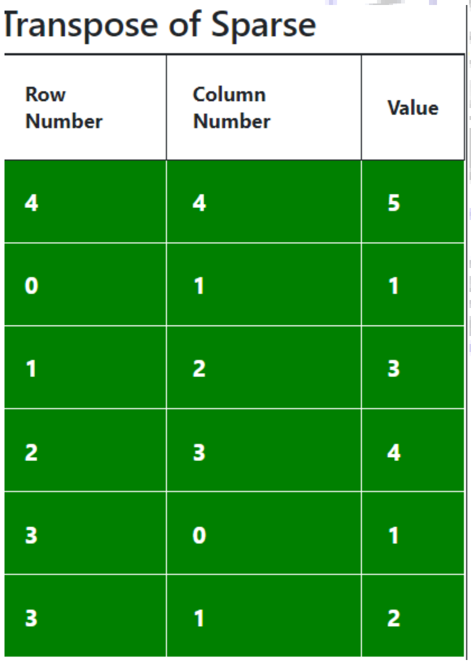


Sparse Storage

Row Number	Column Number	Value
4	4	5
0	3	1
1	0	1
1	3	2
2	1	3
3	2	4

First row corresponds to number of rows, number of columns and number of non-zero elements. Rest of the rows are filled row-wise and in each row column-wise. Only non-zero elements are considered and for each non-zero element, row index, column index and value is recorded.

Using this sparse storage, transpose can be computed by copying from first entry of sparse storage with row and column values being swapped. Then from subsequent entries in sparse storage, lowest column number is found. For each such entry, row and column indices are swapped, value is retained in an entry created in transpose. For the example discussed above, corresponding transpose is:

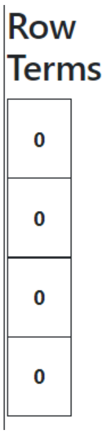


Transpose of Sparse

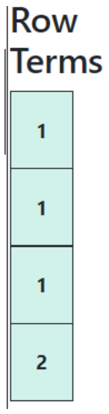
Row Number	Column Number	Value
4	4	5
0	1	1
1	2	3
2	3	4
3	0	1
3	1	2

This transpose computation is inefficient as the complexity = (number of terms)*(number of columns). This multiplicative increase is due to the nested for loops used while computing transpose.

Alternative fast transpose technique:
Row terms array is constructed where number of entries equal to number of columns. Row terms give frequency count of each column index in sparse storage. Row terms is initially initialzed to 0. Later based on the frequency of occurrence of each column index, increment is done.



Row Terms
0
0
0
0

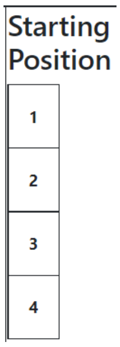


Row Terms
1
1
1
2

Initially After frequency count updation

Starting position of each column index is recorded using formula:
startingpos[index]=startingpos[index-1]+rowterms[index-1]
startingpos[0] is set to 1.

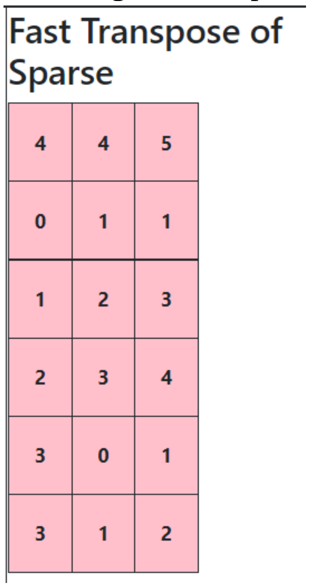
The resulting starting position of the above example is:



Starting Position
1
2
3
4

Fast transpose is computed by finding position of each column index using startingpos array. Then row and column values are swapped and value entry of column index in sparse storage is copied to a position in the fast transpose array indicated by startingpos value.

Resulting fast transpose for the above example is:



Fast Transpose of Sparse		
4	4	5
0	1	1
1	2	3
2	3	4
3	0	1
3	1	2

The complexity of this transpose is = 2*(numterms+numofcolumns). Hence it is an additive increase and do not have any nested loops. This transpose is thus faster than the conventional transpose computation.

Implementation of sparse matrix, transpose and fast transpose:

```
typedef struct
{
    int row;
    int col;
    int value;
} term;

void read_matrix(int mat[10][10],int m,int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&mat[i][j]);
        }
    }
}
void print_matrix(int mat[10][10],int m,int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
          printf("%d  ",mat[i][j]);
        }
        printf("\n");
    }
}

int create_sparse(int mat[10][10],int m,int n, term a[10])
{
    int nzi=0,i,j;
    a[0].row=m;
    a[0].col=n;
    int cnt=1;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            if(mat[i][j]!=0)
            {
                a[cnt].row=i;
                a[cnt].col=j;
                a[cnt].value=mat[i][j];
                cnt++;
            }
        }
    }
    a[0].value=cnt-1;
    return (cnt-1);
}
void display_sparse(term a[10], int nzi)
{
    int i;
    for(i=0;i<=nzi;i++)
    {
        printf("%d  %d   %d\n",a[i].row,a[i].col,a[i].value);
    }

}

void transpose_sparse(term a[20],term b[20])
{
    int i,j,cb;
    b[0].row=a[0].col;
    b[0].col=a[0].row;
    b[0].value=a[0].value;
    cb=1;
    for(i=0;i<a[0].col;i++)  //Column indices
    {
        for(j=1;j<=a[0].value;j++)
        {
            if(a[j].col==i)
            {
                b[cb].row=a[j].col;
                b[cb].col=a[j].row;
                b[cb].value=a[j].value;
                cb++;
            }
        }
    }

}

void fast_transpose(term a[10],term b[10])
{
    int i,j;
    int rowTerms[10],startingPos[10];
    //Fill the first row of transpose
    b[0].row=a[0].col;
    b[0].col=a[0].row;
    b[0].value=a[0].value;
    //Initialize rowterms array elements to 0
    for(i=0;i<a[0].col;i++)
       rowTerms[i]=0;
    //For each column in sparse, increment rowterm entry by 1
    for(j=1;j<=a[0].value;j++)
    {
       rowTerms[a[j].col]++;
    }
    //Initialize statring position first element to 1
    startingPos[0]=1;
    //Update starting position by forumla: sp[i]=sp[i-1]+rowterms[i-1]
    for(i=1;i<a[0].col;i++)
    {
       startingPos[i]=startingPos[i-1]+rowTerms[i-1];
    }
    //For each column in sparse, find its location in transpose
    for(i=1;i<=a[0].value;i++)
    {
       j=startingPos[a[i].col];
       b[j].row=a[i].col;
       b[j].col=a[i].row;
       b[j].value=a[i].value;
       j++;
    }

}
void main()
{
    int mat[10][10],m,n,nzi;
    term a[10],b[10];
    m=3,n=4;
    read_matrix(mat,m,n);
    print_matrix(mat,m,n);
    nzi=create_sparse(mat,m,n,a);
    display_sparse(a,nzi);
    transpose_sparse(a,b);
    printf("TRanspose.....\n");
    display_sparse(b,nzi);
    fast_transpose(a,b);
    printf("Fast TRanspose.....\n");
    display_sparse(b,nzi);

}
```

Module-2

Stacks

Stacks are a special kind of data structure in which insertion and deletion happens at the same end called top of the stack. Stack follows a Last-In-First-Out data structure. Books arranged on top of each other, plates arranged in a reception, box of pringles potato chips and many others are real time examples of the stack.

Stack Operations:

1. CREATE – used to create an empty stack of specific size.
2. PUSH – used to insert an element to the top of the stack
3. POP – used to remove an element from the top of the stack.
4. DISPLAY – display all the elements of the stack.

Stack States:

Stack can enter a state called "Overflow" when one tries to push an element to the stack beyond its capacity. Stack can also enter a state called "Underflow" when one tries to pop an element from an empty stack.

Stacks are static, if the size is fixed during compile time and once stack reaches this size, more elements cannot be pushed. The memory for such a stack is allocated in the stack area of the memory.

Stacks can also be dynamic. Memory for such a stacks is allocated in the heap area of the memory. Once size of the stack reaches max size, memory is doubled and reallocated.

Implementation of static stacks:

```
#define MAXSIZE 5
typedef struct
{
    int data;
}stack;
int top=-1;
int isFull(stack arr[MAXSIZE])
{
    if(top==MAXSIZE-1)
    {
        return 1; //Stack is full
    }
    return 0; //Stack is not full
}

int isEmpty(stack arr[MAXSIZE])
{
    if(top==-1)
    {
        return 1; //Stack is empty
    }
    return 0; //Stack is not empty
}

void push(stack arr[MAXSIZE],int ele)
{
    if(isFull(arr))
    {
        printf("Stack overflow\n");
        return;
    }
    arr[++top].data=ele;
}

void pop(stack arr[MAXSIZE])
{
    if(isEmpty(arr))
    {
        printf("Stack underflow\n");
        return;
    }
    printf("Element popped=%d\n",arr[top].data);
    top--;
}
void display(stack arr[MAXSIZE])
{
    int i;
    if(isEmpty(arr))
    {
        printf("Stack empty\n");
        return;
    }
    for(i=top;i>=0;i--)
    {
        printf("%d\n",arr[i].data);
    }
}

void main()
{
    stack arr[MAXSIZE];
    push(arr,10);
    push(arr,20);
    push(arr,30);
    display(arr);
     push(arr,40);
    push(arr,50);
     push(arr,60);
    push(arr,70);
    display(arr);
    pop(arr);
    pop(arr);
    pop(arr);
    pop(arr);
    pop(arr);
    pop(arr);
}
```

Implementation of dynamic stacks:

```
typedef struct
{
   int ele;
}stack;
int MAXSIZE=3;
int top=-1;
void isFull(stack *arr)
{
   if(top==(MAXSIZE-1))
   {
      MAXSIZE=2*MAXSIZE;
      printf("Stack overflow size increased to %d\n",MAXSIZE);
      arr=realloc(arr,sizeof(stack)*MAXSIZE);
   }

}

int isEmpty()
{
   if(top==-1)
      return 1;
   return 0;
}
void push(stack *arr,int ele)
{
   isFull(arr);
   arr[++top].ele=ele;
}
void pop(stack *arr)
{
   if(isEmpty())
   {
      printf("Stack Underflow..\n");
      return;
   }
   printf("Popped Element=%d\n",arr[top].ele);
```

```
    top--;
}
void display(stack *arr)
{
    int i;
    if(isEmpty(arr))
    {
        printf("Stack Empty..\n");
        return;
    }
    for(i=top;i>=0;i--)
    {
        printf("%d\n",arr[i].ele);
    }
}

void main()
{
    stack *arr;
    arr=malloc(sizeof(stack)*MAXSIZE);
    push(arr,10);
    push(arr,20);
    push(arr,30);
    push(arr,40);
    push(arr,50);
    push(arr,60);
     push(arr,70);
      push(arr,50);
    push(arr,60);
     push(arr,70);
    pop(arr);
    pop(arr);
    display(arr);
}
```

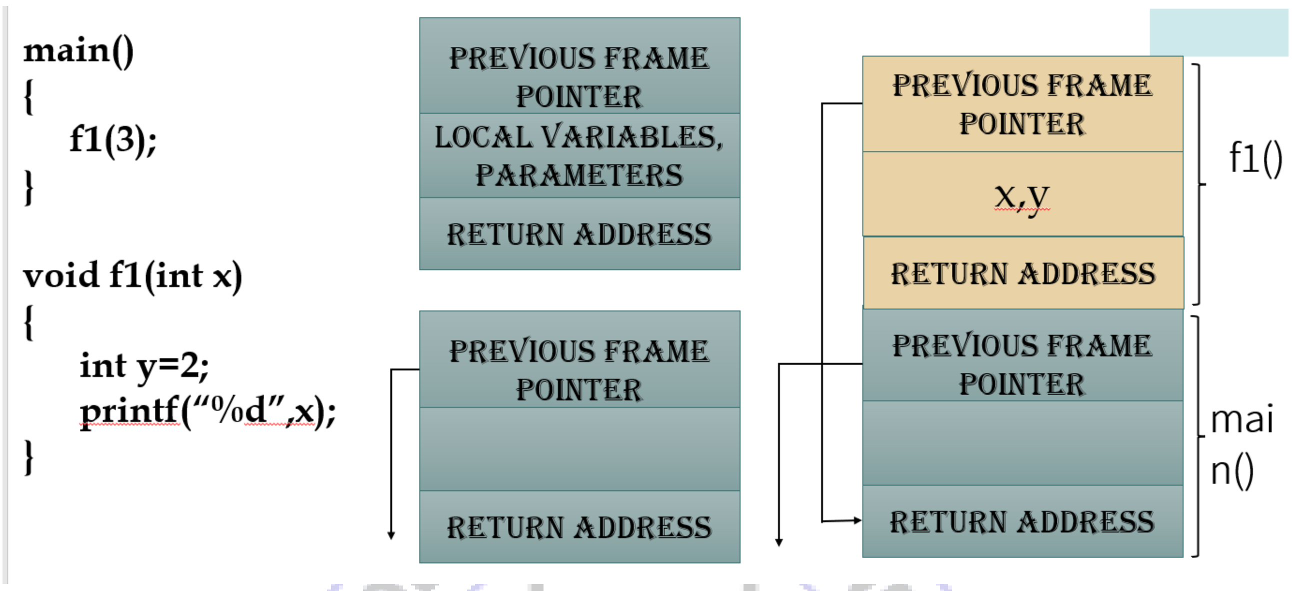
System Stack:

- A special stack used by programs to process function calls.
- Each function is allotted an area called “Activation Record” or “Stack frame”
- Only one function can execute at any time. Hence function to be selected is one at the top of stack
- PUSH- whenever a function is called, new stack frame is pushed on top of stack
- POP – whenever a function execution is completed, stack frame from top is popped

Working of System Stack:
As shown in the following example, 2 functions are called- main and f1. First main function is called, and for main an activation record (stack frame) is allocated. The activation record consists of:

- previous frame pointer which points to address from where it was called and it has to report back

- local variables – variables declared inside the function
- parameters- inputs passed to function
- return address – each function’s return address.



```
main()
{
    f1(3);
}

void f1(int x)
{
     int y=2;
     printf(“%d”,x);
}
```

Queue:

Queue is a data structure that follows First In First Out Order (FIFO). Elements are inserted at “rear end” and elements are deleted from “front end”. Such a queue is also called Single Ended Queue. There are many applications of queue like:

(i) People waiting in queue for ticket booking
(ii) People waiting in queue to get vaccine
(iii) People waiting in queue for lunch during reception
(iv) Students waiting in a queue to collect library books

There are 2 types of Queue: Static Queue (fixed size) and Dynamic Queue (flexible size).
In Static Queue, max size of the queue (number of elements) is fixed during compilation. When the size limit reaches, “Queue Overflow” state occurs. Similarly, when queue is empty, no element can be deleted. If an attempt to deletion is done, it leads to a state called “Queue Underflow”. Further memory for static queue is allocated in stack.

C Program demonstrating static queue:

```
#define MAXSIZE 5
typedef struct
{
   int data;
}queue;
int rear=-1,front=-1;  //Pointers used in Single Ended Queue
int isFull(queue arr[MAXSIZE])
{
   if(rear==MAXSIZE-1)
   {
        return 1; //Queue is full
    }
    return 0; //Queue is not full
}

int isEmpty(queue arr[MAXSIZE])
{
    if(rear==front)
    {
        return 1; //Queue is empty
    }
    return 0; //Queue is not empty
}

void addq(queue arr[MAXSIZE],int ele)
{
    if(isFull(arr))
    {
        printf("Queue overflow\n");
        return;
    }
    arr[++rear].data=ele;
}

void deleteq(queue arr[MAXSIZE])
{
    if(isEmpty(arr))
    {
        printf("Queue underflow\n");
        return;
    }
    printf("Element popped=%d\n",arr[++front].data);

}
void display(queue arr[MAXSIZE])
{
    int i;
    if(isEmpty(arr))
    {
        printf("Queue empty\n");
        return;
    }
    for(i=front+1;i<=rear;i++)
    {
        printf("%d\n",arr[i].data);
    }
}

void main()
{
    queue arr[MAXSIZE];
    addq(arr,20);
    addq(arr,30);
   addq(arr,50);
    addq(arr,60);
     addq(arr,120);
    addq(arr,130);
    display(arr);
    deleteq(arr);
    deleteq(arr);
    deleteq(arr);


}
```

Limitations of Single Ended Queue:

- Rear pointer is incremented during element insertion and front pointer is incremented during element deletion. Hence once all elements are deleted rear and front become same which leads to both overflow and underflow. Hence even though queue is empty, no more elements can be added.
- Possible solutions to such a problem include, shifting of elements during each deletion or resetting front and rear, when they become equal during element deletion. Instead of this an efficient mechanism is to use Circular Queue.
-

Circular Queue:

In Circular Queue, after last element is added, new insertions can begin again from beginning, provided deletion of some elements are done. This solves the problems of single ended queue. To implement circular queue, modulus addition is used for both rear and front, instead of simple increment. Further, a counter is used for tracking number of elements in the queue at any time. Counter is incremented, as a new element is added to the circular queue. Once the count reaches maximum size, "Queue Overflow" occurs. Counter is also decremented when element is deleted from queue. Once the count reduces to -1, "Queue Underflow" occurs.

C Program for implementation of Circular Queue:

```
#define MAXSIZE 5
typedef struct
{
    int data;
} queue;
int front=-1,rear=-1,cnt=0;
int isFull(queue arr[MAXSIZE])
{
    if(cnt==MAXSIZE)
        return 1;//Queue overflow
    return 0;//Queue still free
}

int isEmpty(queue arr[MAXSIZE])
{
    if(cnt==0)
```

```
        return 1; //Empty
    return 0; //Non Empty
}

void addq(queue arr[MAXSIZE], int ele)
{
    if(isFull(arr))
    {
        printf("Queue overflow\n");
        return;
    }
    rear=(rear+1)%MAXSIZE;
    arr[rear].data=ele;
    cnt++;
}

void deleteq(queue arr[MAXSIZE])
{
    if(isEmpty(arr))
    {
        printf("Queue underflow\n");
        return;
    }
    front=(front+1)%MAXSIZE;
    printf("Element deleted=%d\n",arr[front].data);
    cnt--;

}
void display(queue arr[MAXSIZE])
{
    int i,j;
    j=front+1;
    for(i=0;i<cnt;i++)
    {
        printf("%d\n",arr[j].data);
        j=(j+1)%MAXSIZE;
    }
}
void main()
{
    queue arr[MAXSIZE];
    addq(arr,20);
    addq(arr,30);
    addq(arr,40);
    display(arr);
    addq(arr,50);
    addq(arr,60);
    addq(arr,70);
    display(arr);
    deleteq(arr);
    deleteq(arr);
    // display(arr);
    addq(arr,100);
    addq(arr,110);
    addq(arr,120);
    display(arr);
}
```

Circular Queue using Dynamic Arrays (Dynamic Circular Queues):

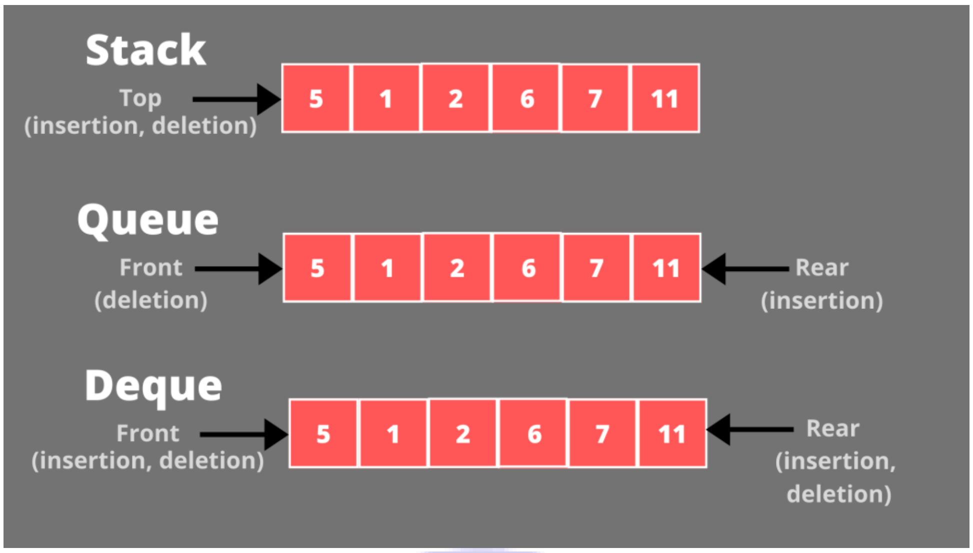
Static Circular Queues have the issues of Overflow whenever number of elements in the queue equals the max size of the queue. Further, static circular queues use stack which has limited memory. Hence, dynamic circular queues are used, where memory is allocated in the heap and size doubles each time limit is reached. Dynamic memory allocation function malloc is used to create a circular queue with some maximum size. Once limit is reached, realloc function is used to double the current size.

C Program to implement Dynamic Circular Queues:

```
int MAXSIZE=5;
typedef struct
{
    int data;
} queue;
int front=-1,rear=-1,cnt=0;
void isFull(queue *arr)
{
    if(cnt==MAXSIZE)
     {
        MAXSIZE=2*MAXSIZE;
        arr=realloc(arr,sizeof(queue)*MAXSIZE);
     }
}

int isEmpty(queue *arr)
{
    if(cnt==0)
        return 1; //Empty
    return 0; //Non Empty
}

void addq(queue *arr, int ele)
{
    isFull(arr);
      rear=(rear+1)%MAXSIZE;
    arr[rear].data=ele;
    cnt++;
}

void deleteq(queue *arr)
{
    if(isEmpty(arr))
    {
        printf("Queue underflow\n");
        return;
    }
    front=(front+1)%MAXSIZE;
    printf("Element deleted=%d\n",arr[front].data);
    cnt--;

}
void display(queue *arr)
{
    int i,j;
    j=front+1;
    for(i=0;i<cnt;i++)
    {
        printf("%d\n",arr[j].data);
        j=(j+1)%MAXSIZE;
    }
}
void main()
{
    //queue arr[MAXSIZE];
    queue *arr=malloc(sizeof(queue)*MAXSIZE);
    addq(arr,20);
    addq(arr,30);
    addq(arr,40);
    display(arr);
    addq(arr,50);
    addq(arr,60);
    addq(arr,70);
    display(arr);
    deleteq(arr);
    deleteq(arr);
    // display(arr);
    addq(arr,100);
    addq(arr,110);
    addq(arr,120);
    display(arr);
}
```

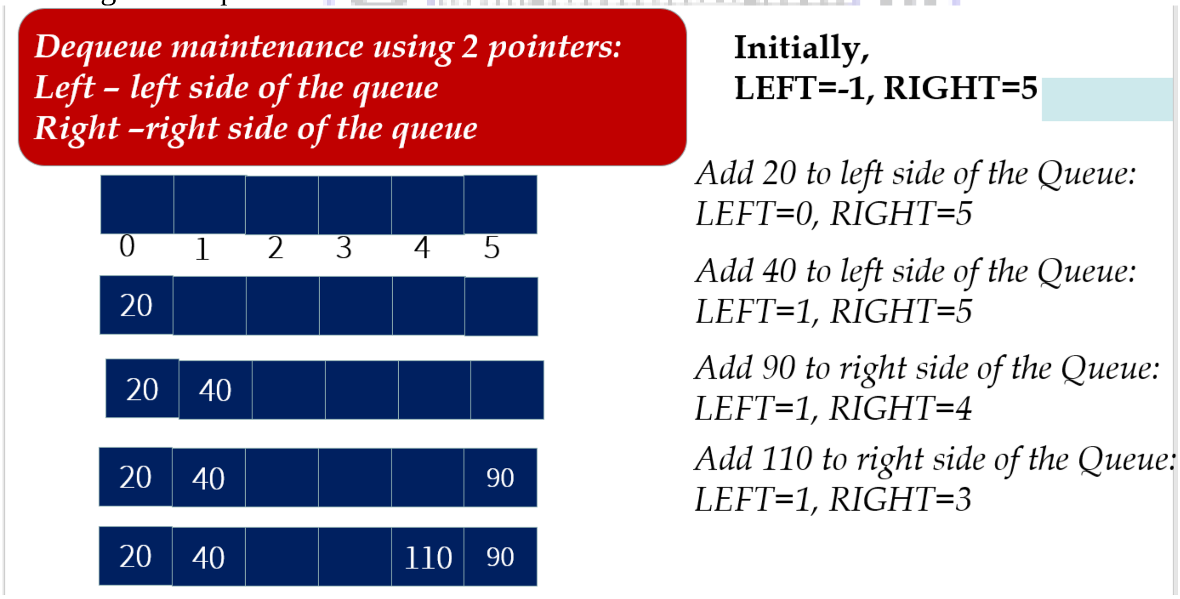
Dequeue (Deck or Double Ended Queue):

Dequeue is a data structure that supports addition and deletion of elements at either ends (front/rear). Hence it uses 2 pointers left and right side of the queue. When elements are inserted to left end, left pointer is incremented (++left) and when elements are inserted to the right end, right pointer is decremented (right--). Relationship between stack, single ended/circular queue and dequeue is as follows:



Browser history is a memory maintained by each browser. Whenever user browses a page, it is added to the history. Pages can be added to both ends and also removed from either end. Hence double ended queue is used for browser history management.

Working of a dequeue is illustrated below:


Dequeue maintenance using 2 pointers:
Left – left side of the queue
Right –right side of the queue

Initially,
LEFT=-1, RIGHT=5

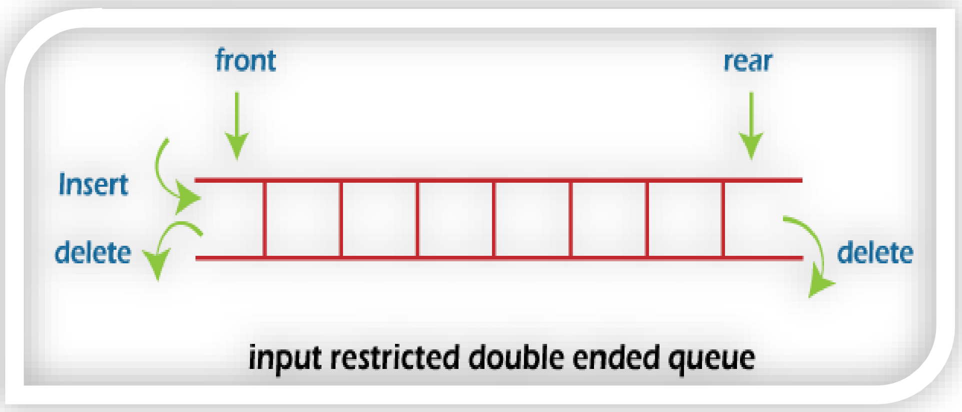
Add 20 to left side of the Queue:
LEFT=0, RIGHT=5

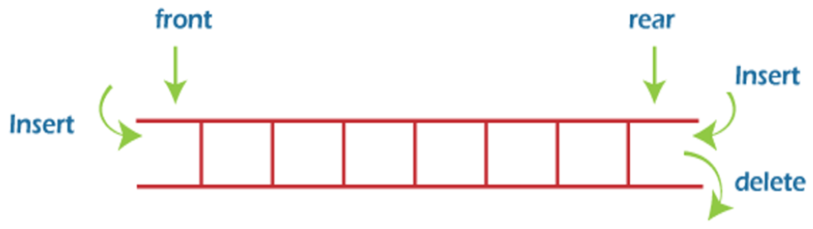
Add 40 to left side of the Queue:
LEFT=1, RIGHT=5

Add 90 to right side of the Queue:
LEFT=1, RIGHT=4

Add 110 to right side of the Queue:
LEFT=1, RIGHT=3

Variations of Dequeue:
There are two variations of dequeue: (i) Input Restricted dequeue (Input-Rear end no insertion only, deletion, but at front end both insertion and deletion can be done)
(ii) Output Restricted dequeue: (Output- front end no deletion, only insertion. However insertion and deletion can be done at rear end)





Output restricted double ended queue

Priority Queues:

A Priority Queue is a collection of elements with each element assigned a priority. Deletion of an element is based on following rules:

1. An element of higher priority before lower priority
2. For same priority elements, FIFO

Timesharing systems with high priority programs first and programs with normal priority into standard queue

There are 2 ways to represent priority queues in memory:

(i) As a Linear List.
(ii) Using Arrays

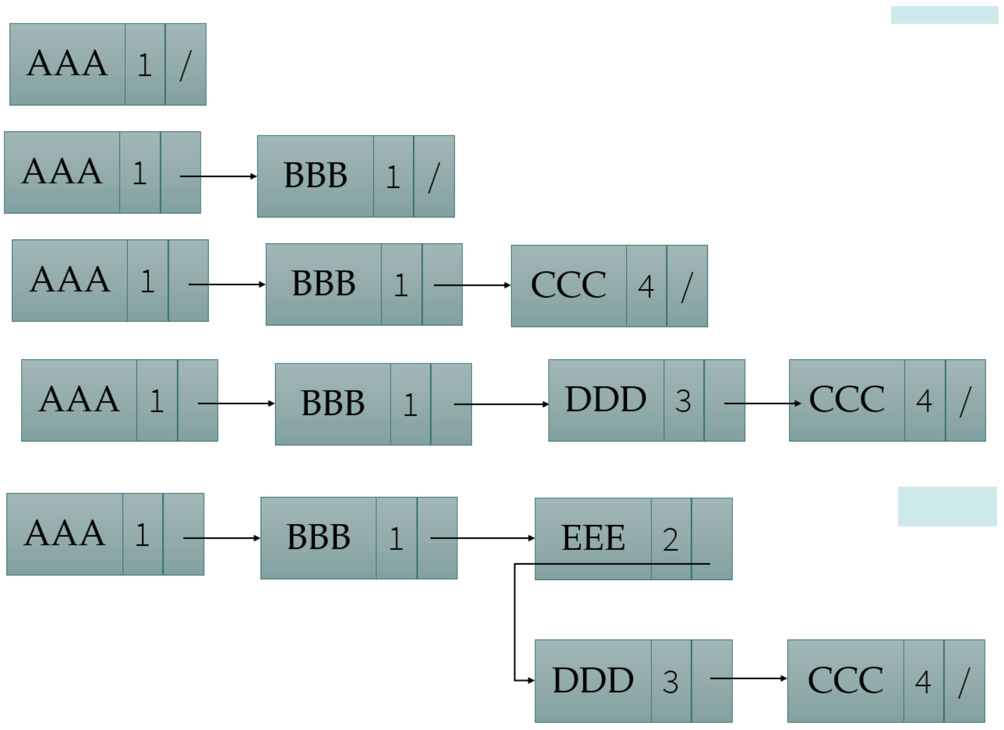
In Linear List, it is a collection of nodes connected with link. Each node contains INFO (Information/data), PRN (Priority Number) and Link (pointer to the next node). For last node, pointer is null.

- Rule: Lower numbers indicate higher priority
- Higher priority nodes are behind lower priority
- Same priority nodes are arranged in the order added

Insertion:
For insertion search for a node whose priority is greater than this node

Deletion:
Deletion is always done by removing first node

Eg:
(AAA,1), (BBB,1), (CCC,4), (DDD,3), (EEE,2)



Hence with linear list, elements need to be shuffled each time, a new element of higher priority is to be added. Instead of shuffling elements again and again, another solution is to use multiple queues. A separate queue is maintained for each priority level. Separate front and rear pointers are maintained for each such queue. The front and rear pointers are captured in separate arrays. Arrays for front and rear, contains position of front and rear for each queue represented by the priority level. Further, these arrays are used to construct a matrix . Rows of the matrix represents priority level and columns contain the elements of the queue.
Eg:

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Here, consider row 1, front is 2 and rear is 2. Hence element currently is AAA. In case new element is to be added to row 1(Queue with priority 1), it will be added at position 3. Similarly for row 2, new element will be added at position 4 (rear is 3) and element will be deleted from position 1 (front is 1).

Algorithm for deletion:

1. *Find smallest K such that FRONT[K]!=NULL*
2. *Delete and process front element in row K of Queue*
3. *Exit*

Algorithm for Insertion of an ITEM with priority M:

1. *Insert ITEM to rear element in row M of Queue*
2. *Exit*

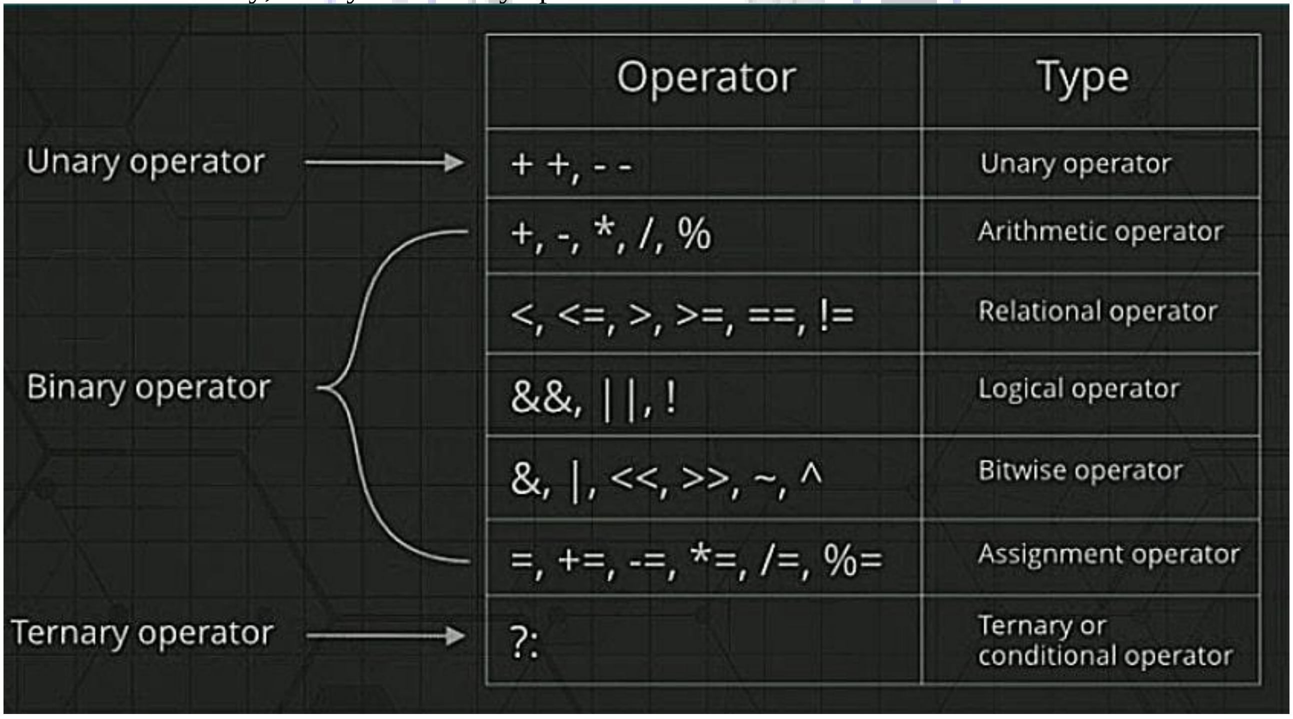
Stack Applications – Expressions

Expressions contain operators and operands. Arithmetic expressions are evaluated. While evaluation, BEDMAS rule is used. Precedence of operator determines which sub expression is first evaluated. In case of same precedence, associativity determines the order of evaluation (Left to right or Right to Left).

Eg: Evaluate the expression: x=a/b-c+d*e-a*c, where a=4, b=c=2, d=e=3

```
x=a/b-c+d*e-a*c
  =4/2-2+3*3-4*2
  =2-2+3*3-4*2
  =2-2+9-4*2
  =2-2+9-8
  =2+7-8
   =9-8
   =1
```

Precedence of unary, binary and ternary operators are shown below:



	Operator	Type
Unary operator	+ +, - -	Unary operator
Binary operator	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&, | |, !	Logical operator
	&, |, <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator	?:	Ternary or conditional operator

If number of operands is 1, operator is unary. Likewise, if operands is 2, operator is binary and for 3 operands, operator is ternary.

Types of Expressions:

There are 3 types of expressions based on the position of operator. If operator is inbetween operands, expression is called "Infix Expression". If operator comes before operands, expression is "Prefix" and if operator comes after operands, expression is "Postfix". Prefix expressions are also called

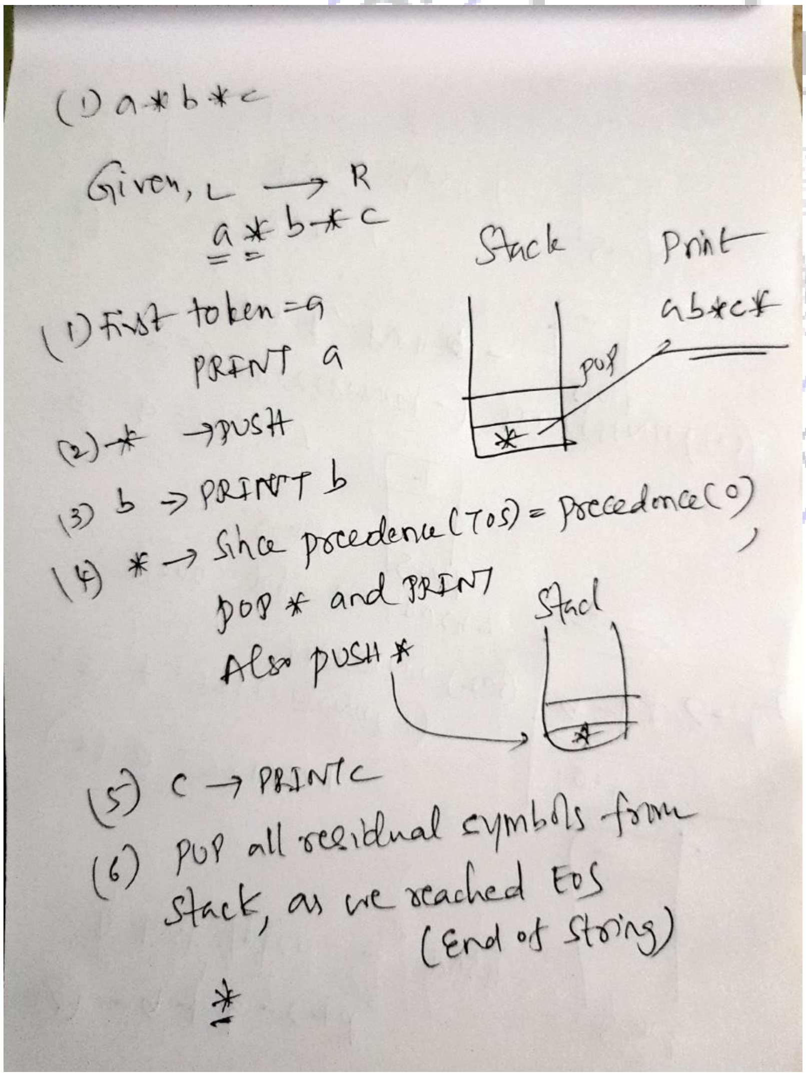
"Polish" and postfix expressions are also called "Reverse Polish or suffix". Postfix and Prefix expressions are used by compilers due to efficiency concerns during evaluation.

Algorithm for converting Infix to Postfix:

```
STEP 1: SCAN EXPRESSION FROM LEFT TO RIGHT
STEP 2: IF CURRENT TOKEN IS NUMBER/CHAR, PRINT IT. GOTO STEP 1
STEP 3: IF CURRENT TOKEN IS OPERATOR O:
    3.1: IF PRECEDENCE(O) <= PRECEDENCE (TOS)
        POP TOS SYMBOL and PRINT
        REPEAT 3.1
    3.2: ELSE PUSH O to STACK
        GOTO STEP 1
STEP 4: IF CURRENT TOKEN IS '(', PUSH TO STACK. GO TO STEP 1
STEP 5: IF CURRENT TOKEN IS ')', POP AND PRINT ALL SYMBOLS UNTIL YOU
GET '('. GOTO STEP 1
STEP 6: POP ANY RESIDUAL SYMBOLS AND PRINT
STEP 7: IF CURRENT TOKEN IS EOS, EXIT
```

Eg:

$\boldsymbol{a * b * c}$


Stack
Print
ab*c*
pop
*
Stack
*

(1) a*b*c

Given, L → R

$\underline{\underline{a*}}$b*c

(1) First token = a
PRINT a

(2) * → PUSH

(3) b → PRINT b

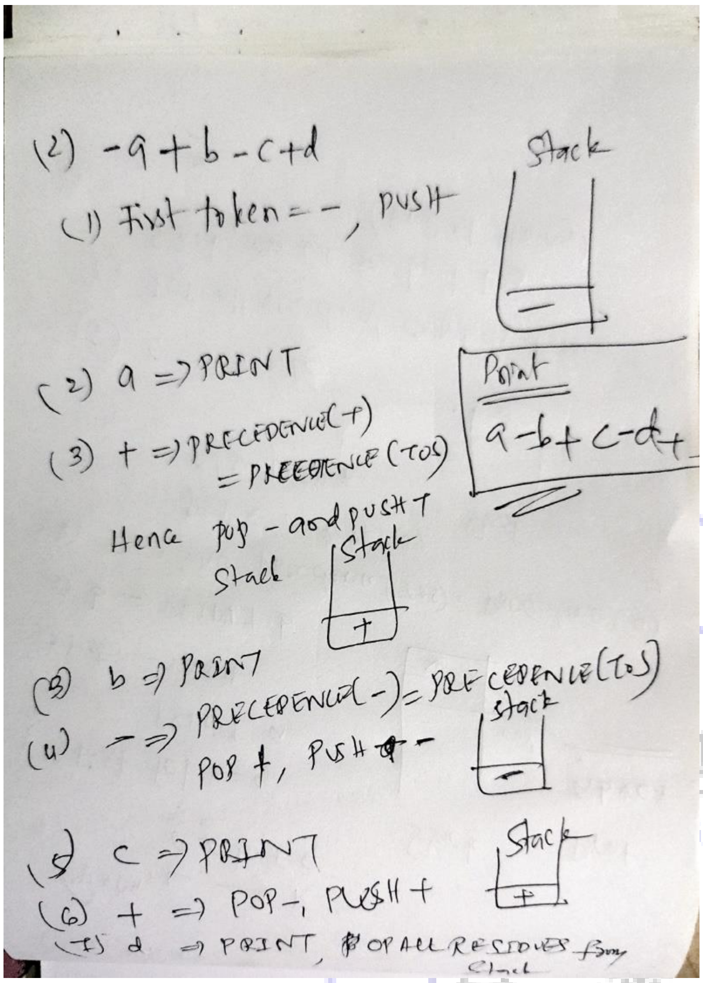
(4) * → Since precedence(TOS) = Precedence(O),
pop * and PRINT
Also PUSH *

(5) c → PRINT c

(6) POP all residual symbols from stack, as we reached EOS
(End of String)

$\underline{*}$

$-a + b - c + d$



(2) -a + b - c + d

(1) First token = -, push

Stack

(2) a ⇒ PRINT

(3) + ⇒ PRECEDENCE(+) = PRECEDENCE (TOS)

Hence pop - and push +

Stack

Print

a - b + c - d +

(3) b ⇒ PRINT

(4) - ⇒ PRECEDENCE(-) = PRECEDENCE(TOS)

POP +, PUSH -

stack

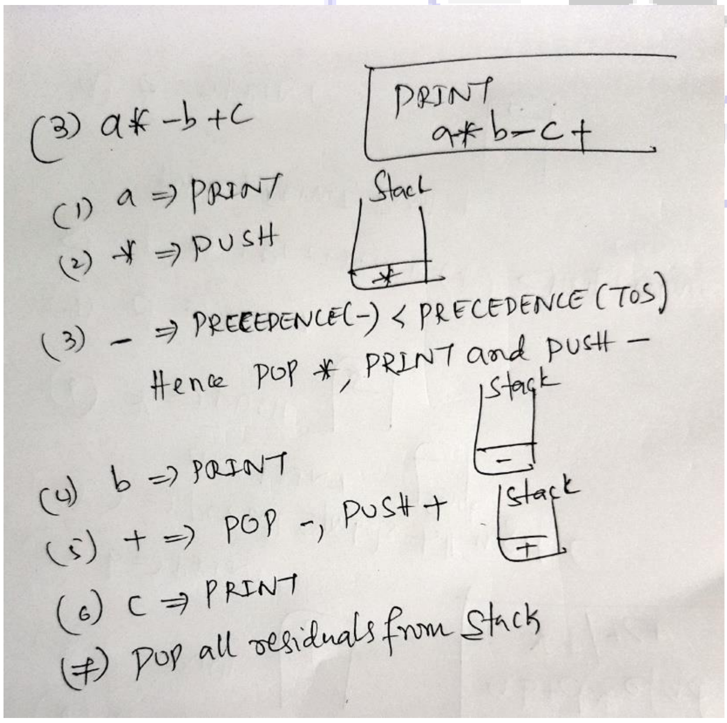
(5) c ⇒ PRINT

(6) + ⇒ POP -, PUSH +

Stack

(7) d ⇒ PRINT, POP ALL RESIDUES from Stack

$a * -b + c$



(3) a * -b + c

PRINT

a * b - c +

(1) a ⇒ PRINT

(2) * ⇒ PUSH

Stack

(3) - ⇒ PRECEDENCE(-) < PRECEDENCE (TOS)

Hence POP *, PRINT and PUSH -

Stack

(4) b ⇒ PRINT

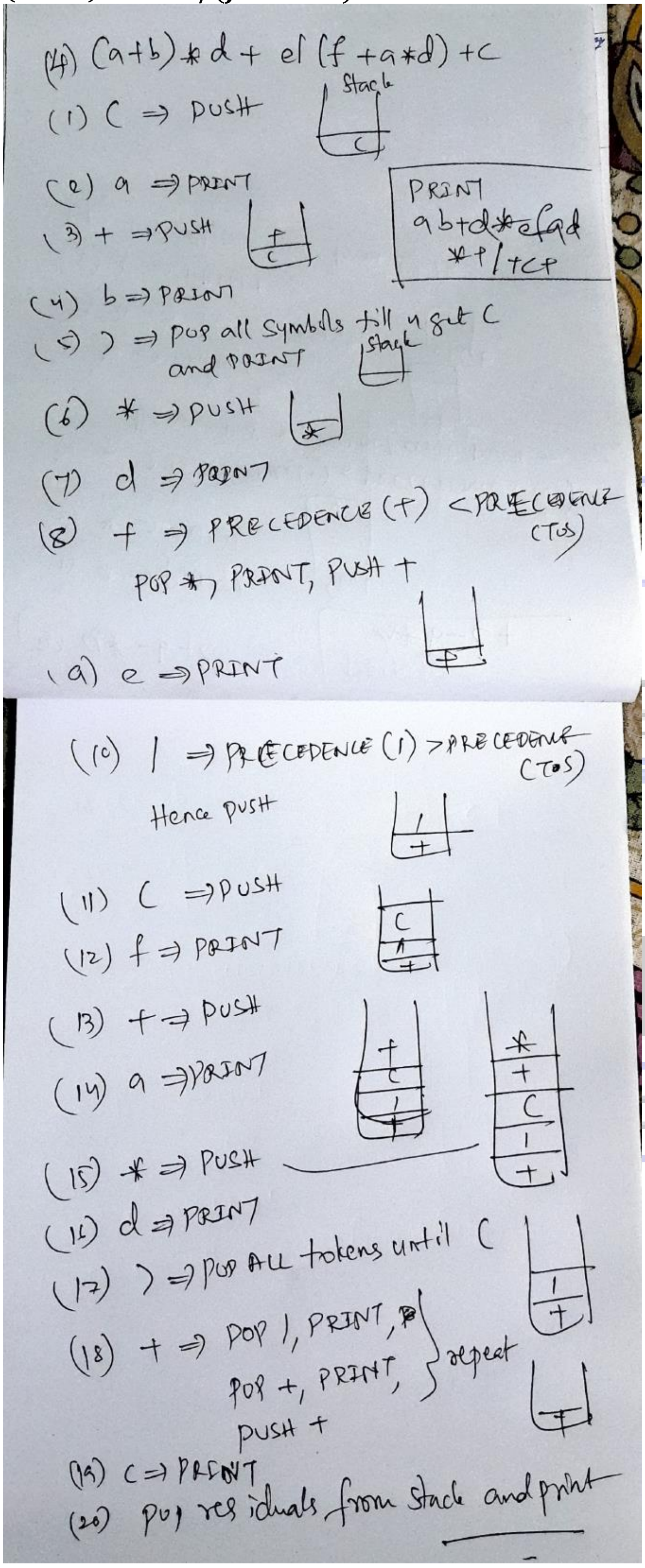
(5) + ⇒ POP -, PUSH +

Stack

(6) c ⇒ PRINT

(7) Pop all residuals from Stack

$(a + b) * d + e/(f + a * d)$ +c



(4) (a+b)*d+ e/(f+a*d)+c

(1) (⇒ PUSH

(2) a ⇒ PRINT

(3) + ⇒ PUSH

PRINT
ab+d*efad
*+/+c+

(4) b ⇒ PRINT

(5)) ⇒ POP all Symbols till u get (
and PRINT

(6) * ⇒ PUSH

(7) d ⇒ PRINT

(8) + ⇒ PRECEDENCE (+) < PRECEDENCE (TOS)
POP *, PRINT, PUSH +

(9) e ⇒ PRINT

(10) / ⇒ PRECEDENCE (/) > PRECEDENCE (TOS)
Hence PUSH

(11) (⇒ PUSH

(12) f ⇒ PRINT

(13) + ⇒ PUSH

(14) a ⇒ PRINT

(15) * ⇒ PUSH

(16) d ⇒ PRINT

(17)) ⇒ POP ALL tokens until (

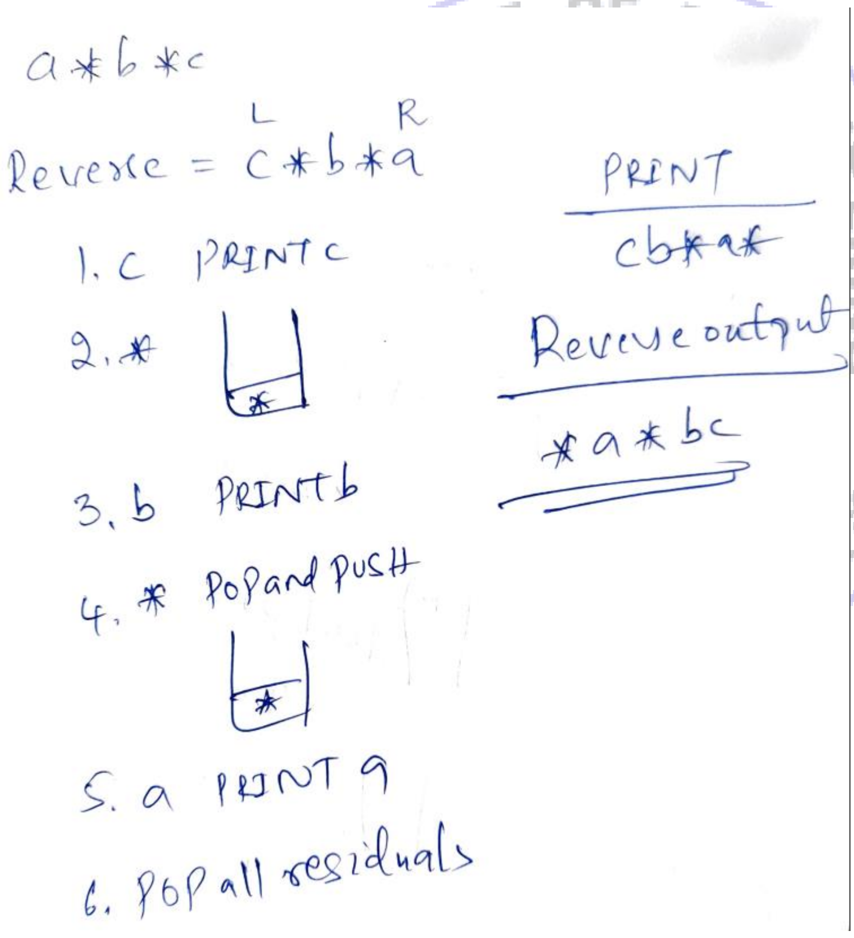
(18) + ⇒ POP /, PRINT, } repeat
POP +, PRINT,
PUSH +

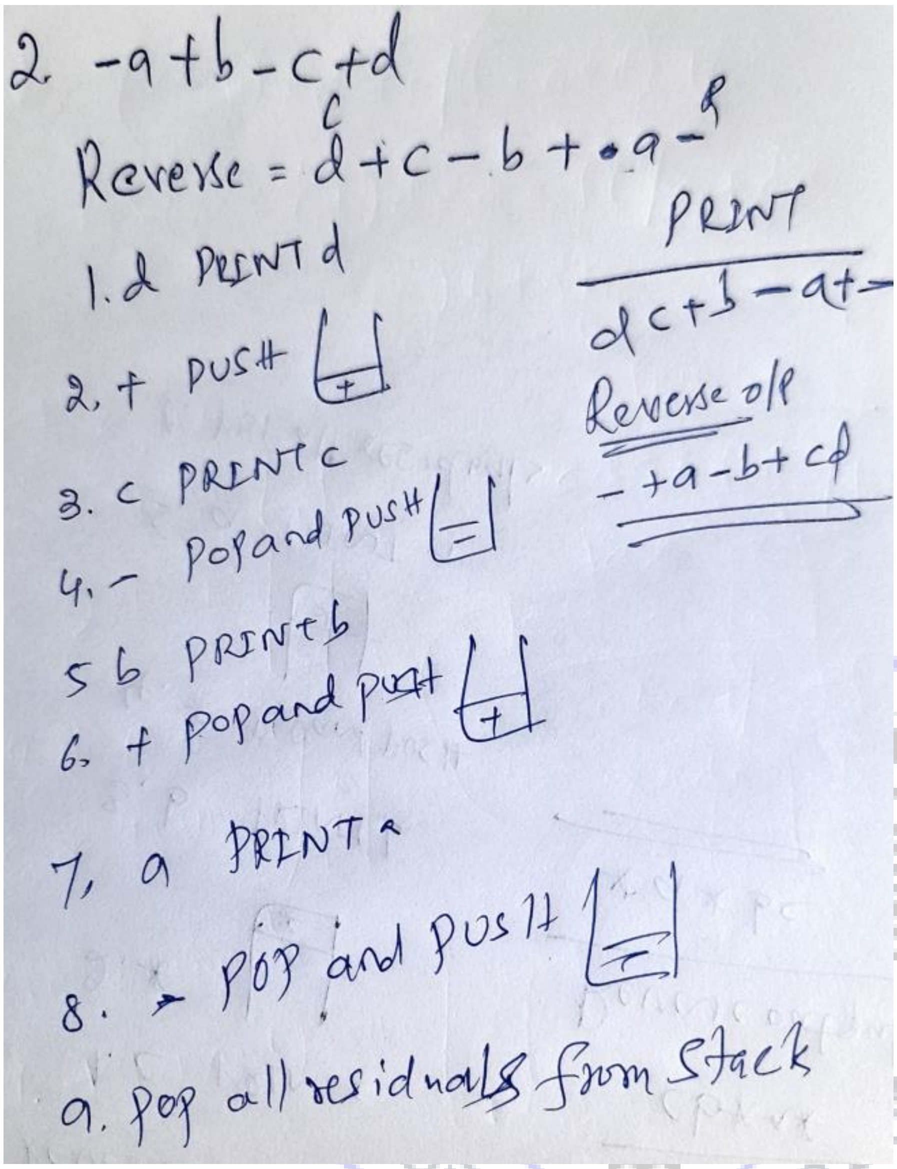
(19) c ⇒ PRINT

(20) POP residuals from stack and print

Algorithm for converting Infix to Prefix:

```
STEP 1: REVERSE THE GIVEN EXPRESSION
STEP 2: SCAN EXPRESSION FROM LEFT TO RIGHT
STEP 3: IF CURRENT TOKEN IS NUMBER/CHAR, PRINT IT. GOTO STEP 1
STEP 4: IF CURRENT TOKEN IS OPERATOR O:
        4.1: IF PRECEDENCE(O) <= PRECEDENCE (TOS)
                POP TOS SYMBOL and PRINT
                REPEAT 4.1
        4.2: ELSE PUSH O to STACK
                GOTO STEP 1
STEP 5: IF CURRENT TOKEN IS ‘)‘, PUSH TO STACK. GO TO STEP 1
STEP 6: IF CURRENT TOKEN IS ‘(’,  POP AND  PRINT ALL SYMBOLS UNTIL YOU
              GET ‘)‘. GOTO STEP 1
STEP 7: POP ANY RESIDUAL SYMBOLS AND PRINT
STEP 8: IF CURRENT TOKEN IS EOS, REVERSE OUTPUT and  EXIT
```





2. $-a+b-c+d$

Reverse = $d+c-b+a-$

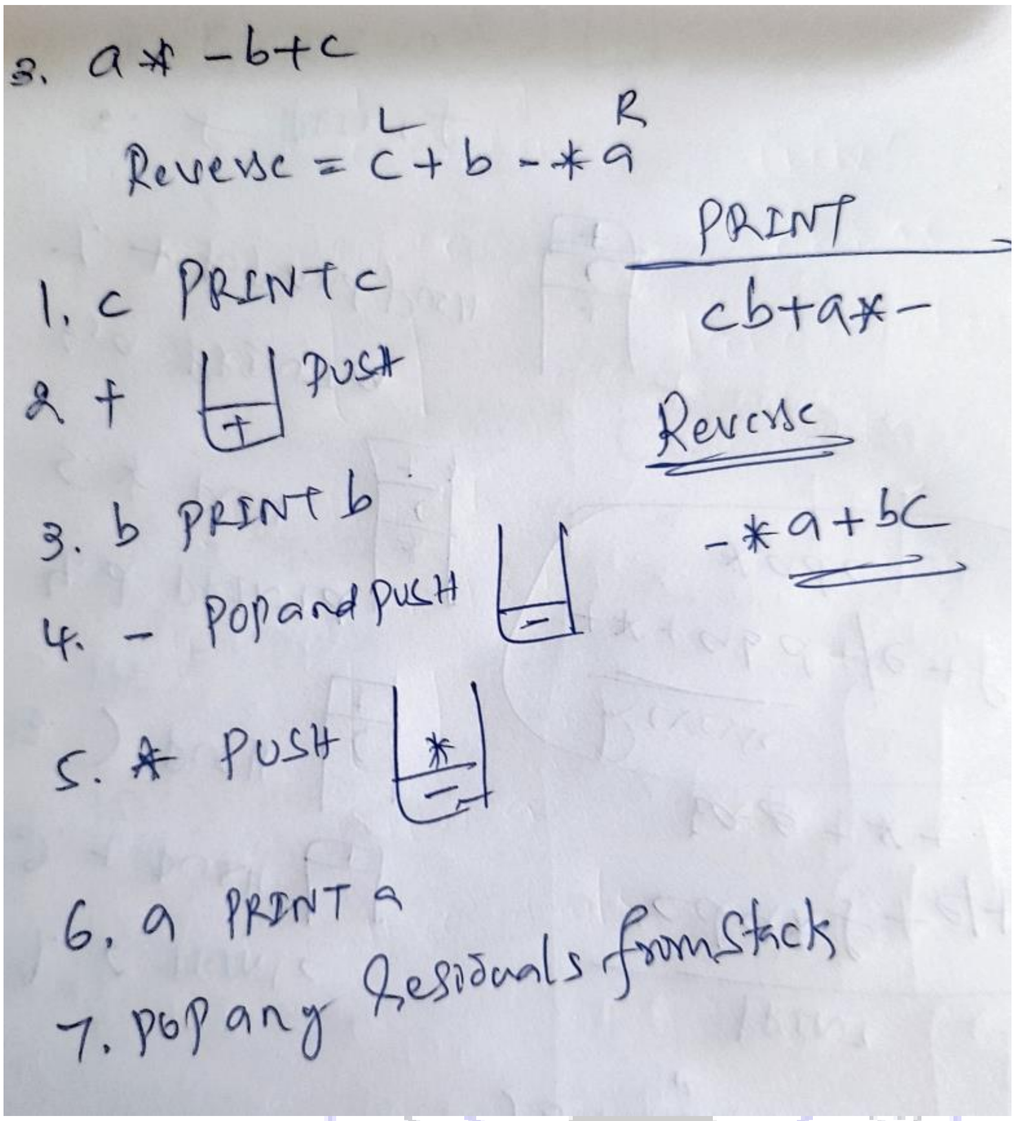
1. d PRINT d
2. + PUSH $\boxed{+}$
3. c PRINT c
4. − Pop and PUSH $\boxed{-}$
5. b PRINT b
6. + Pop and push $\boxed{+}$
7. a PRINT a
8. − POP and PUSH $\boxed{-}$
9. Pop all residuals from Stack

PRINT

$dc+b-a+-$

Reverse o/p

$-+a-b+cd$



3. a* -b+c

Reverse = $\overset{L}{c}$+b-*$\overset{R}{a}$

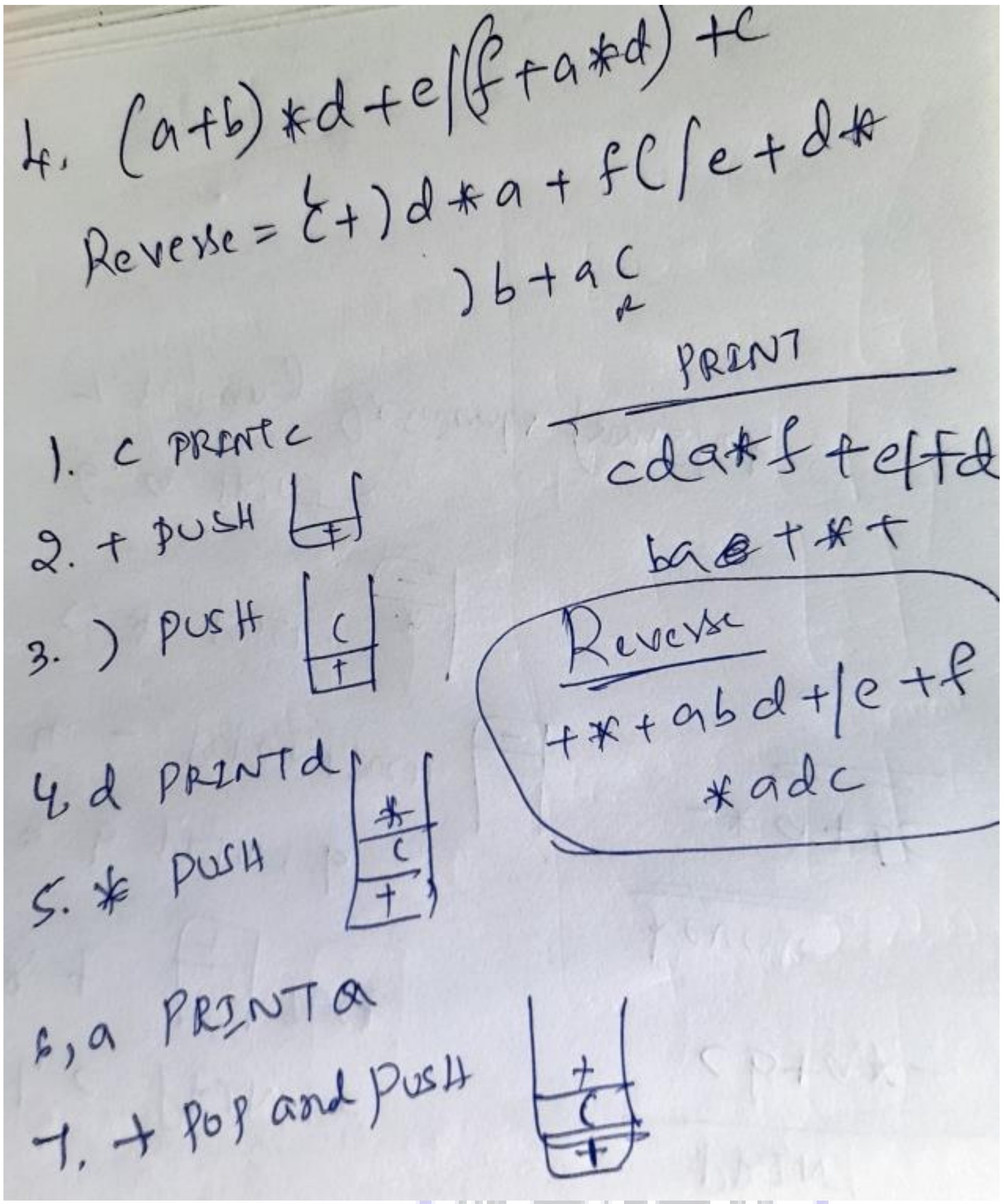
1. c PRINT c
2. + PUSH
3. b PRINT b
4. - Pop and PUSH
5. * PUSH
6. a PRINT a
7. Pop any Residuals from Stack

PRINT

cb+a*-

Reverse

-*a+bc



4. (a+b)*d+e/(f+a*d)+c

Reverse = c+)d*a+f(/e+d*
)b+a(

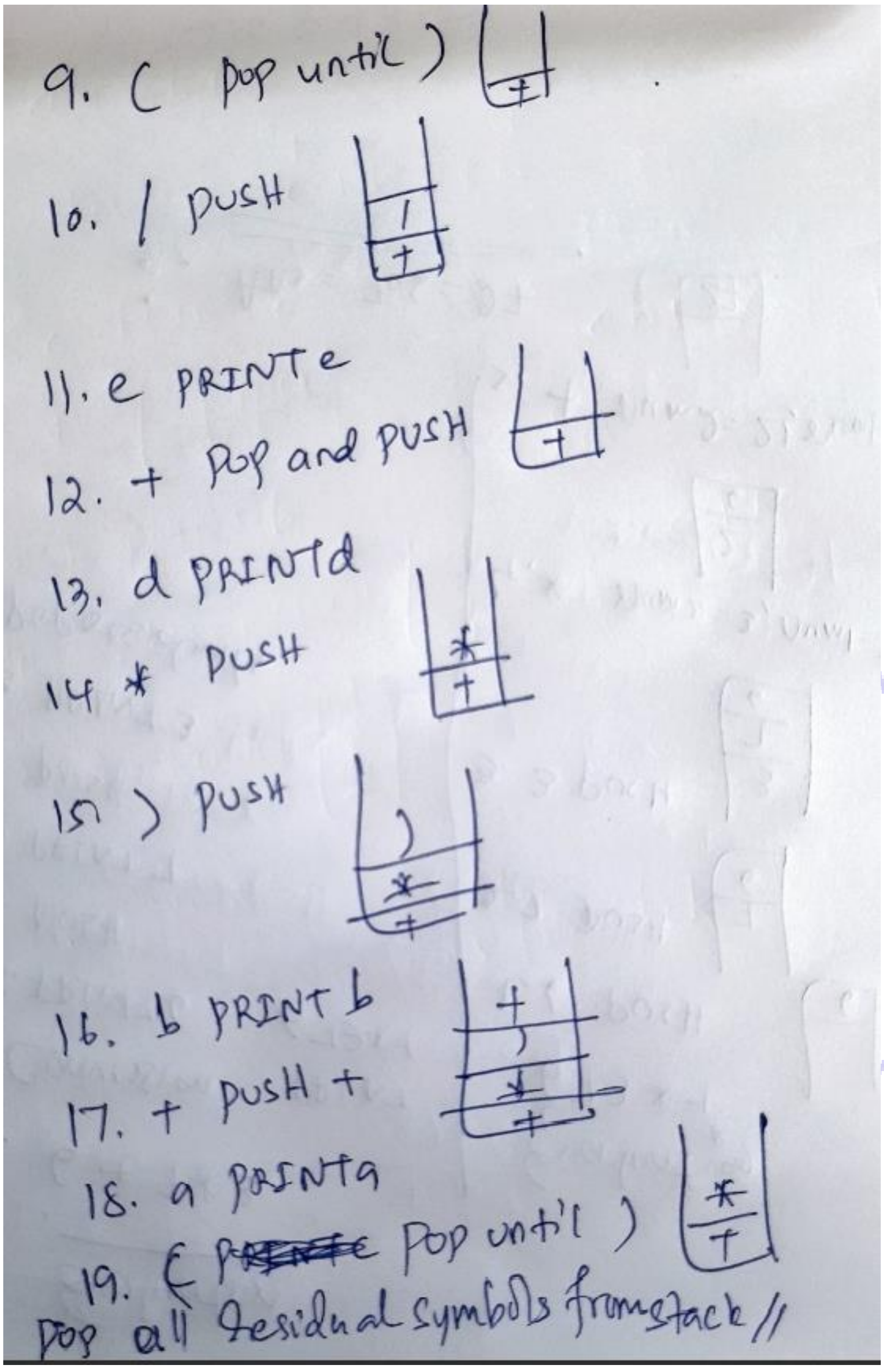
1. c PRINT c
2. + PUSH
3.) PUSH
4. d PRINT d
5. * PUSH
6, a PRINT a
7. + POP and PUSH

PRINT

cda*f+e/fd
ba+*+

Reverse

+*+abd+/e+f
*adc



9. (Pop until)

10. / push

11. e PRINT e

12. + Pop and PUSH

13. d PRINT d

14 * PUSH

15) PUSH

16. b PRINT b

17. + PUSH +

18. a PRINT a

19. (~~PRINT~~ Pop until)

Pop all residual symbols from stack //

Algorithm for the Evaluation of Postfix Expressions:

ALGORTITHM:

STEP 1: SCAN EXPRESSION FROM LEFT TO RIGHT

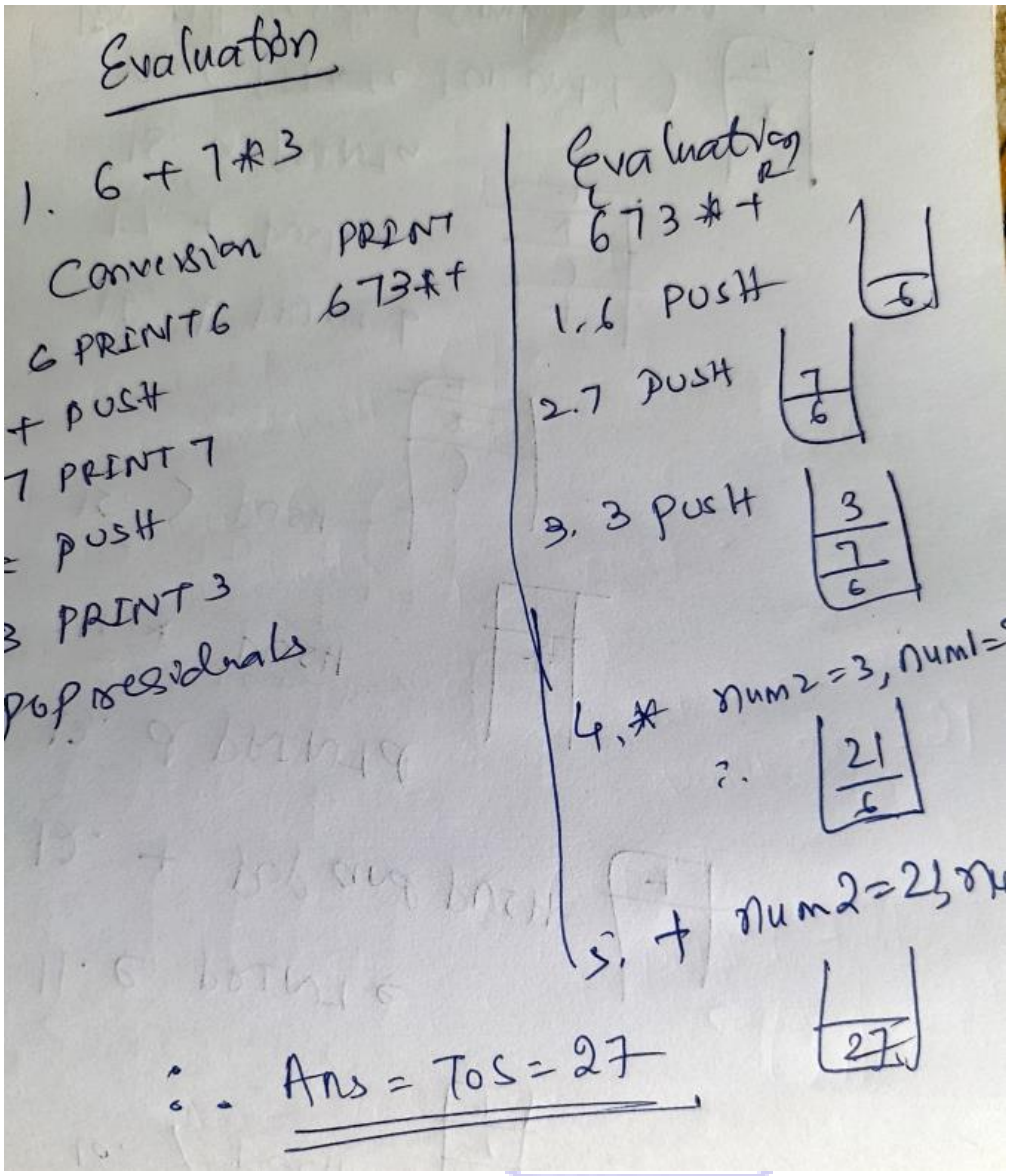
STEP 2: IF CURRENT TOKEN IS NUMBER/CHAR, PUSH NUMBER/CHAR. GOTO STEP 1

STEP 3: IF CURRENT TOKEN IS OPERATOR O:

```
        3.1: POP TOS SYMBOL –NUM2
             POP TOS SYMBOL – NUM1
             PUSH (NUM1 OP NUM2)
```

GOTO STEP 1

STEP 4: IF CURRENT TOKEN IS EOS, PRINT TOS and EXIT



Evaluation

1. 6 + 7*3

Conversion PRINT

673*+

6 PRINT 6

+ push

7 PRINT 7

* push

3 PRINT 3

pop residuals

Evaluation

673*+

1. 6 push

6

2. 7 push

7
6

3. 3 push

3
7
6

4. * num2=3, num1=

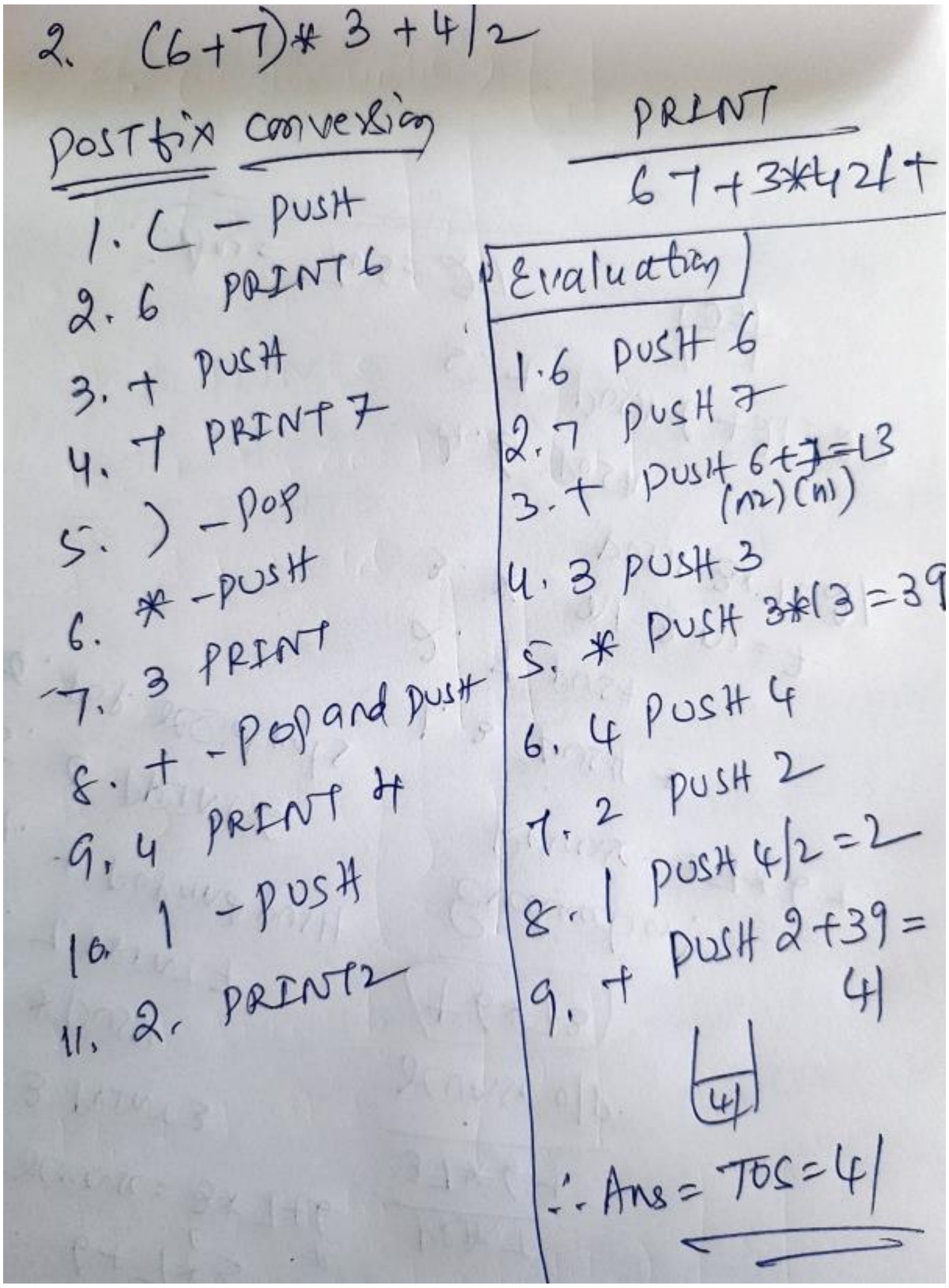
∴

21
6

5. + num2=21, nu

27

∴ Ans = TOS = 27



2. (6+7)*3+4/2

Postfix conversion

1. (– push
2. 6 PRINT 6
3. + push
4. 7 PRINT 7
5.) – pop
6. * – push
7. 3 PRINT
8. + – pop and push
9. 4 PRINT 4
10. / – push
11. 2. PRINT 2

PRINT

67+3*42/+

Evaluation

1. 6 push 6
2. 7 push 7
3. + push 6+7=13 (n2)(n1)
4. 3 push 3
5. * push 3*13=39
6. 4 push 4
7. 2 push 2
8. / push 4/2=2
9. + push 2+39=41

41

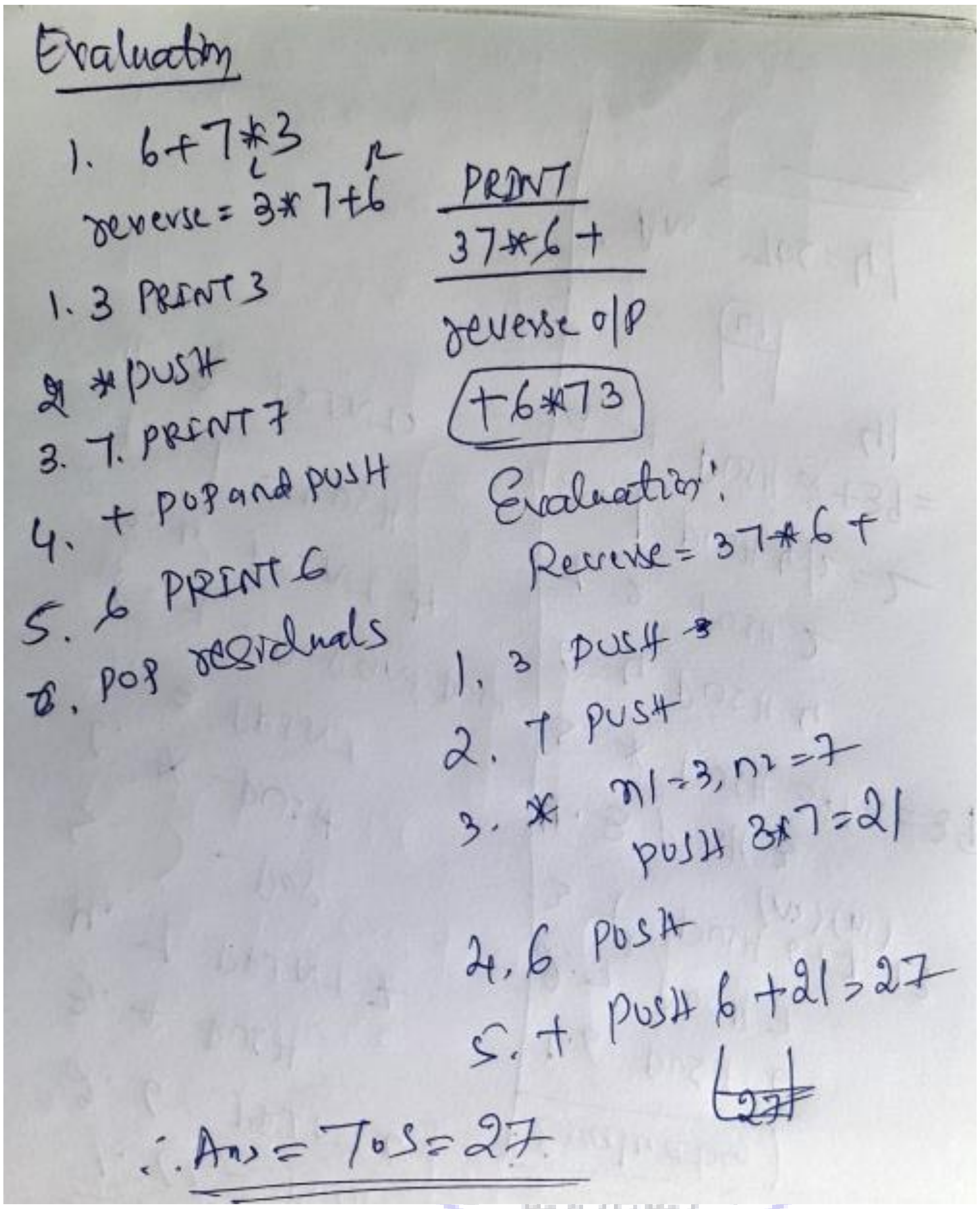
∴ Ans = TOS = 41

C program for Evaluation:

```
#define MAX 20
typedef struct
{
    int data;
}stack;
int top=-1;
void push(stack arr[MAX],int ele)
{
    arr[++top].data=ele;
}
int pop(stack arr[MAX])
{
   return arr[top--].data;
}
void main()
{
   char expr[MAX];
   stack arr[MAX];
   int num1,num2,i;
   printf("Enter the valid postfix expression\n");
   scanf("%s",expr);
   for(i=0;i<strlen(expr);i++)
   {
      if(isdigit(expr[i]))  //digit
      {
         push(arr,expr[i]-'0');
      }
      else
      {
         num2=pop(arr);
         num1=pop(arr);
         switch(expr[i])
         {
            case '+':push(arr,num1+num2);
                      break;
           case '-':push(arr,num1-num2);
                      break;
            case '*':push(arr,num1*num2);
                      break;
            case '/':push(arr,num1/num2);
                      break;
         }
      }
   }
   printf("Ans=%d\n",arr[top].data);
}
```

Algorithm for the Evaluation of Prefix Expressions:

```
ALGORTITHM:
STEP  1: REVERSE THE GIVEN EXPRESSION
STEP 2: SCAN EXPRESSION FROM LEFT TO RIGHT
STEP 3: IF CURRENT TOKEN IS NUMBER/CHAR, PUSH NUMBER/CHAR. GOTO STEP 1
STEP 4: IF CURRENT TOKEN IS OPERATOR O:
      4.1: POP TOS SYMBOL –NUM1
           POP TOS SYMBOL – NUM2
           PUSH (NUM1 OP NUM2)
           GOTO STEP 1
STEP 5: IF CURRENT TOKEN IS EOS, PRINT TOS and EXIT
```



Evaluation

1. $6+7\underset{L}{*}3$ R

reverse = $3*7+6$

1. 3 PRINT 3
2. * PUSH
3. 7. PRINT 7
4. + pop and push
5. 6 PRINT 6
6. pop residuals

PRINT

$37*6+$

reverse o/p

$\boxed{+6*73}$

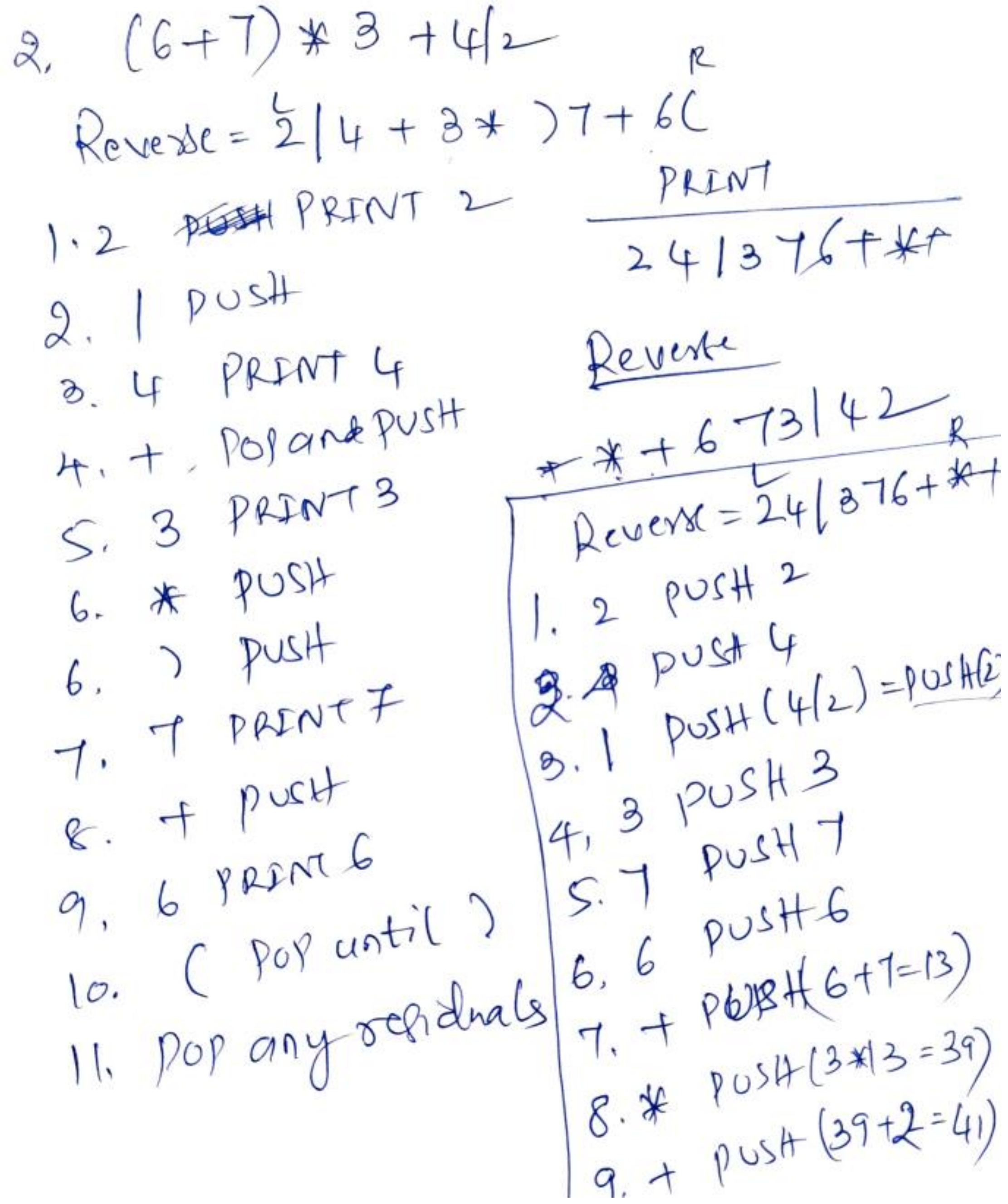
Evaluation:

Reverse = $37*6+$

1. 3 push 3
2. 7 push
3. * $n1=3, n2=7$
   push $3*7=21$
4. 6 push
5. + push $6+21=27$

$\boxed{27}$

$\therefore$ Ans = TOS = 27



2. $(6+7)*3+4/2$

Reverse = $\overset{L}{2}/4+3*)7+6\overset{R}{(}$

1. 2 ~~PUSH~~ PRINT 2
2. / PUSH
3. 4 PRINT 4
4. + Pop and PUSH
5. 3 PRINT 3
6. * PUSH
6.) PUSH
7. 7 PRINT 7
8. + push
9. 6 PRINT 6
10. (Pop until)
11. Pop any residuals

PRINT

24/376+*+

Reverse

+*+673/42

Reverse = $\overset{L}{2}4/376+*\overset{R}{+}$

1. 2 push 2
2. 4 push 4
3. / push (4/2) = push(2)
4. 3 push 3
5. 7 push 7
6. 6 push 6
7. + push (6+7=13)
8. * push (3*13=39)
9. + push (39+2=41)

Recursion:

Recursion is the process of function calling itself. Recursive functions are written using recurrence relations. Recursion is ended by specifying base criteria. A stack frame is created for each call during recursive function.

Eg: Finding factorial of a number using recursion:
Recurrence relation:

$$\boldsymbol{fact(n)} = \begin{cases} \boldsymbol{n = 1} & \boldsymbol{1} \\ \boldsymbol{n > 1} & \boldsymbol{n * fact(n-1)} \end{cases}$$

C code for factorial using recursion:

```
int  fact(int n)
```

```
{
   if(n==1) return 1;
   else
      return n*fact(n-1);
}
void main()
{
   int n=5;
   printf("%d\n",fact(n));
}
```

Eg: Finding n terms of Fibonacci series using recursion:

Recurrence relation:

$$fib(n) = \begin{cases} n = 0 \text{ or } n = 1 & n \\ n > 1 & fib(n-1) + fib(n-2) \end{cases}$$

C code for Fibonacci series using recursion:

```
int  fib(int n)
{
   if(n==0 || n==1)
      return n;
   else
      return fib(n-1)+fib(n-2);
}
void main()
{
   int n=5,i;
   for(i=0;i<n;i++)
      printf("%d,",fib(i));
}
```

Eg: Finding gcd of 2 numbers using recursion:
GCD of 2 numbers using following recurrence relation:

$$gcd(m,n) = \begin{cases} n \neq 0 & gcd(n, m\%n) \\ else & m) \end{cases}$$

C Program:
```
int gcd(int m, int n)
{
   if(n!=0)
      return gcd(n,m%n);
    else
      return m;

}
void main()
{
   int m=81,n=27;
   int ans=gcd(m,n);
   printf("%d\n",ans);
}
```

Tower of Hanoi:

It is a mathematical game or puzzle consisting of three legs and a number of disks of various diameters, which can slide onto any leg. The puzzle begins with the disks stacked on one leg in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last leg, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty leg.
3. No disk may be placed on top of a disk that is smaller than it.

Recursive Solution:
1. MOVE TOP N-1 DISKS FROM LEG A to LEG B
2. MOVE THE TOP DISK FROM LEG A TO LEG C
3. MOVE TOP N-1 DISKS FROM LEG B TO LEG C

C program:
```
void toh(int N,char A,char B, char C)
{
   if(N>0) //termination condition (base condition)
   {
   toh(N-1,A, C, B);
   printf("%c->%c\n",A,C);
   toh(N-1,B,A,C);
   }
}
void main()
{
   int N=3;
   toh(N,'A','B','C');
}
```

Module-3

Linked Lists:

A linked list or one way list is a linear collection of nodes. Each node contains data and a pointer next that points to the next element in the list. Addition, deletion, updation, traversal, searching, sorting, merging frequently needed on the linked list.

Operations on linked lists:

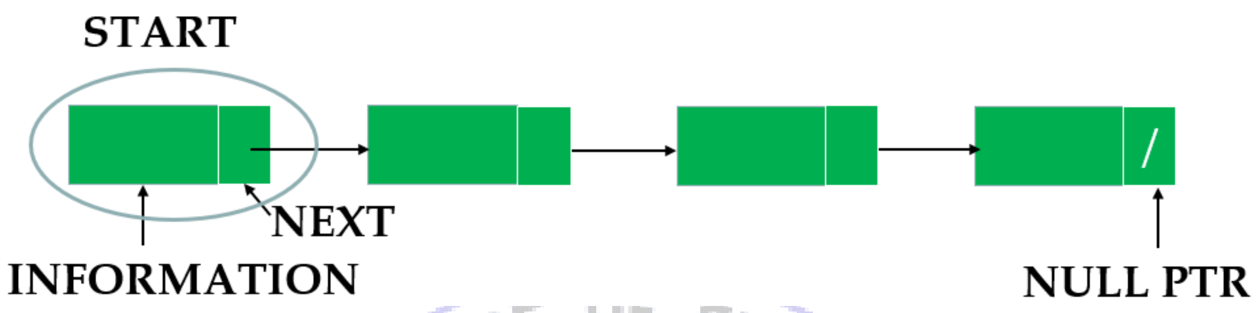
- Addition – an operation to add a new element to a list. Possible positions include beginning, ending, specific and depending on sort requirement.
- Deletion – an operation to remove an existing element from the list. Deletion can be done on any node in the list.
- Updation – an operation to update an existing element in the list. Updation can be done on any node in the list.
- Traversal – an operation in which nodes are visited from beginning till end one by one.
- Searching – an operation to find an element of interest. Result can be successful or unsuccessful.
- Sorting – an operation to order the linked list by ascending or descending order of the information in the linked list.
- Merging – an operation to copy data from 2 lists into a single list.

Arrays vs Linked Lists:

Parameter	**Array**	**Linked List**
Memory allocation	Arrays need contiguous allocation. Hence as number of elements are large, difficult to find contiguous allocation.	Linked list provide or use allocation which is not contiguous. As the number of elements increases, linked lists are the better solution for the memory allocation of elements.
Memory allocation mechanism	Arrays primarily use static allocation. However dynamic allocation is also possible.	Linked list depends primarily on dynamic allocation. However, static allocation is possible but sparingly used.
Accessing an element	Direct access of any element is possible through indexing.	Indexing is not possible. To access any element traversal is required till desired element is found.
Adding a new element	Needs lot of shifting of elements towards right side	Only pointer adjustment of 2 nodes are required.
Deleting an existing element	Needs lot of shifting of elements towards left side.	Only pointer adjustment of 2 nodes are required.

Memory Issues	Garbage and dangling pointer issues do not occur.	Garbage and dangling pointer issues do occur.
Overheads	No overheads as elements are contiguously stored.	Each node has an overhead to maintain extra pointer.

Schematic of a Linked List:

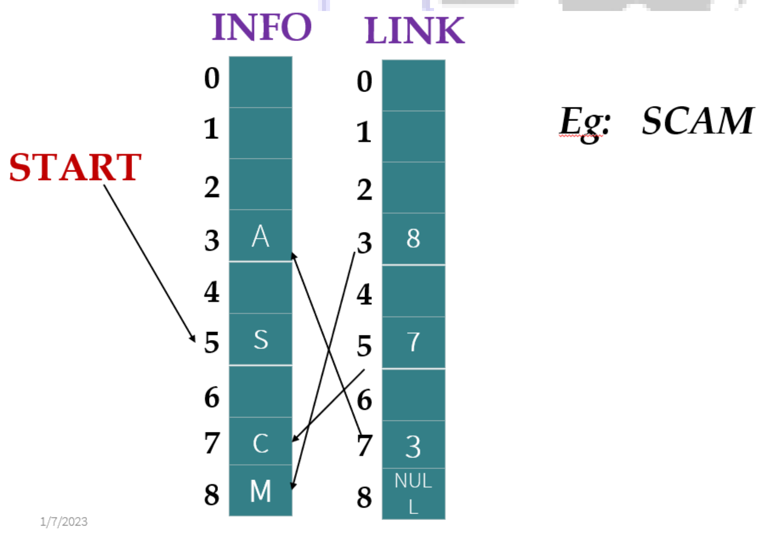


Each node consists of information and a pointer called NEXT. NEXT points to the next element in the list. If the node is the last node, NEXT is a null pointer. (NULL). The beginning node is referred by a special pointer called START.

Example:
Linked list can be used to store patient list admitted to beds. The linked list is sorted in the ascending order of patient names alphabetically.

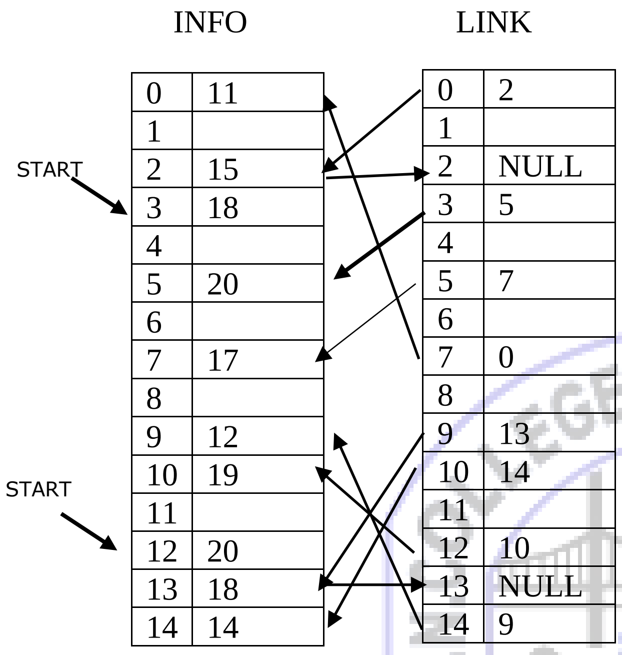
Representation of Linked List in the memory:

Eg-1: Represent the word SCAM in memory:



Two arrays are maintained: INFO and LINK. String is scanned from left to right. For each character, place is allotted in the INFO array wherever free. The next location of each letter is recorded in the corresponding position in the LINK array.

Consider IA scores in subjects:
DS- 18,20,17,11,15
ADE- 20,19,14,12,18
Represent this as 2 linked lists in memory



	INFO
0	11
1	
2	15
3	18
4	
5	20
6	
7	17
8	
9	12
10	19
11	
12	20
13	18
14	14

	LINK
0	2
1	
2	NULL
3	5
4	
5	7
6	
7	0
8	
9	13
10	14
11	
12	10
13	NULL
14	9

Discuss how an employee file containing 3 records with fields (EMPID, Name, Designation, Rank) can be stored using linked lists and without linked lists. Compare and contrast both approaches

An employee file with 3 records and 4 fields can be stored as 3 X 4 2D array. Each row will represent an employee record and each column represent field of employee. This organization will help to access any employee data directly with appropriate row and column indexing. However, when number of employees increase, the contiguous allocation issue arises and fails. Further adding and removing an element in the specific positions would incur lot of penalty with the amount of shifting involved. Hence a linked list representation would solve issues of memory, complexity of addition and deletion operations. For linked list representation, each node represents and employee with information field divided into 4 separate fields for empid, name, designation and rank and a next field to point to the next employee record.

C program to implement basic operations of a linked list:

```
#include<stdlib.h>
typedef struct
{
   int data;
   struct node *next;
}node;
```

```
/** Function to traverse a singly linked list (SLL)
Algorithm:
Step 1: Initialize temp node with START
Step 2: If temp==NULL, goto Step 6
Step 3: Print temp->data
Step 4: Move to next node with temp=temp->next
Step 5: Goto Step 2
Step 6: EXIT
*/
void traverse(node *start)
{
   node*temp=start;
   while(temp!=NULL)
   {
      printf("%d->",temp->data);
      temp=temp->next;
   }
   printf("\b\b  \n");
}
```

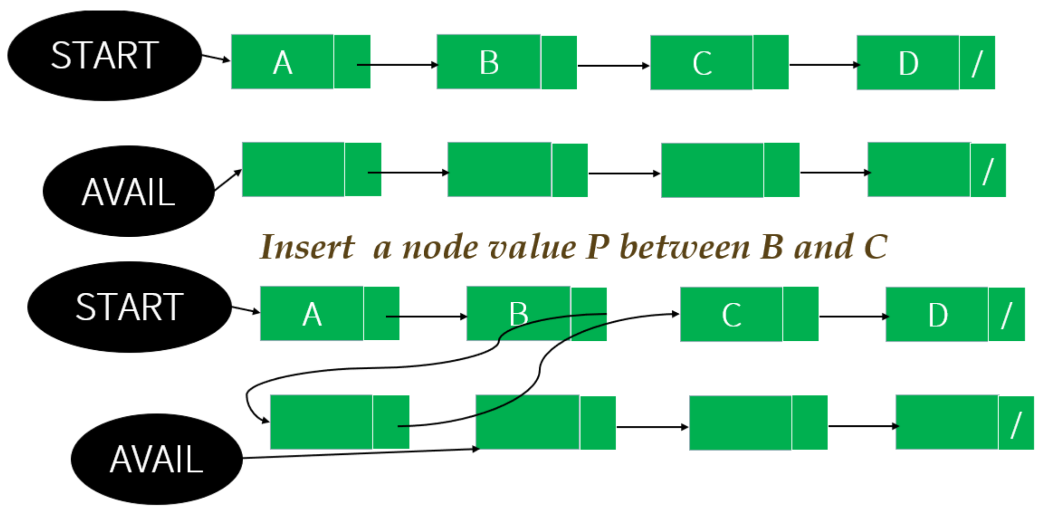
```
/** Function to count the number of nodes of a singly linked list (SLL)
Algorithm:
Step 1: Initialize temp node with START and COUNT to 0.
Step 2: If temp==NULL, goto Step 6
Step 3: COUNT=COUNT+1
Step 4: Move to next node with temp=temp->next
Step 5: Goto Step 2
Step 6: Print COUNT and EXIT
*/

void count_nodes(node *start)
{
   int cnt=0;
   node*temp=start;
   while(temp!=NULL)
   {
      cnt++;
      temp=temp->next;
   }
   printf("cnt=%d \n",cnt);
}
```

```
/** Function to search an element KEY in the unsorted singly linked list (SLL)
Algorithm:
Step 1: Initialize temp node with START
Step 2: If temp==NULL, goto Step 6
Step 3: If  temp->data == KEY, PRINT SUCCESSFUL SEARCH and EXIT
Step 4: Move to next node with temp=temp->next
Step 5: Goto Step 2
Step 6: PRINT UNSUCCESSFULL SEARCH and EXIT
*/

void search_unsorted_list(node *start)
{
   node *temp=start;
   int key,pos=0;
   printf("Enter the key element\n");
   scanf("%d",&key);
   while(temp!=NULL)
   {
      pos++;
      if(temp->data==key)
      {
         printf("Successful search and position=%d\n",pos);
         return;
      }

      temp=temp->next;
   }
   printf("Unsuccessful search\n");

}

/** Function to search an element KEY in the sorted singly linked list (SLL)
Algorithm:
Step 1: Initialize temp node with START
Step 2: If temp==NULL, goto Step 7
Step 3: If  temp->data == KEY, PRINT SUCCESSFUL SEARCH and EXIT
Step 4: If temp->data > KEY, PRINT UNSUCCESSFUL SEARCH and EXIT
Step 5: Move to next node with temp=temp->next
Step 6: Goto Step 2
Step 8: PRINT UNSUCCESSFULL SEARCH and EXIT
*/

void search_sorted_list(node *start)
{
   node *temp=start;
   int key,pos=0;
   printf("Enter the key element\n");
   scanf("%d",&key);
   while(temp!=NULL)
   {
      pos++;
      if(temp->data==key)
      {
         printf("Successful search and position=%d\n",pos);
         return;
      }
```

```
        if(key<temp->data)
        {
             printf("Unsuccessful search\n");
           return;
        }

        temp=temp->next;
    }
    printf("Unsuccessful search\n");

}
void main()
{
    node *n1,*n2,*n3,*n4,*n5;
    n1=malloc(sizeof(node)); //Dynamic memory allocation to create any node
    n2=malloc(sizeof(node));
    n3=malloc(sizeof(node));
    n4=malloc(sizeof(node));
    n5=malloc(sizeof(node));
    n1->data=9;
    n2->data=10;
    n3->data=15;
    n4->data=20;
    n5->data=51;
    n1->next=n2;
    n2->next=n3;
    n3->next=n4;
    n4->next=n5;
    n5->next=NULL;

    traverse(n1);
    count_nodes(n1);
   // search_unsorted_list(n1);
   search_sorted_list(n1);
}
```

Insertion of a new node to the linked list:

Insertion of a new node in the linked list, requires allocating memory to the new node and adjusting pointers appropriately. Two pointers are maintained: START- start of the existing singly linked list and AVAIL- start of free pool. Whenever a new node to be created, first node from free pool is picked by accessing AVAIL and AVAIL is subsequently changed to point next free element. The current node is a node after which a new node has to be inserted. New node is made to point to the node which is pointed by current node. Current node then points to the new node. This mechanism is pictorially represented in the following.



C program to demonstrate different type of insertions:

```
#include<stdlib.h>
typedef struct
{
    int data;
    struct node *next;
}node;

node *start=NULL;


/**Function to insert a given node at the beginning
Algorithm:
STEP 1: Get the data for new node
STEP 2: Allocate memory for new node
STEP 3: Assign data to the new node
STEP 4: New node points to START
STEP 5: New START is set to new node
STEP 6: EXIT
*/
void insert_begining()
{
    int ele;
    node *newnode;
    printf("Enter the element\n");
    scanf("%d",&ele);
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->next=start;
    start=newnode;


}
```

```
/**Function to insert a given node after a given location
Algorithm:
STEP 1: Get the data for new node
STEP 2; Get the required location LOC
STEP 3: Allocate memory for new node
STEP 4: Assign data to the new node
STEP 5: Set temp to START and CNT=1
STEP 6: IF temp==NULL goto STEP 10
STEP 7: IF CNT==LOC, Goto STEP 9
STEP 8: Move temp to next node: temp=temp->next and CNT=CNT+1. Goto STEP 6
STEP 9: Set: newnode->next to temp->next and temp->next to newnode.
STEP 10: EXIT
*/

void insert_after()
{
   int ele,loc,cnt=1;
   node *newnode,*temp;
   printf("Enter the element\n");
   scanf("%d",&ele);
   printf("Enter the location\n");
   scanf("%d",&loc);
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   temp=start;
   while(temp!=NULL)
   {
      if(cnt==loc)
      {
         break;
      }
      else
      {
         temp=temp->next;
         cnt++;
      }

   }
   newnode->next=temp->next;
   temp->next=newnode;

}

/**Function to insert a node in the sorted SLL (Singly Linked List)
Algorithm:
STEP 1: Get the data for new node
STEP 2: Allocate memory for new node
STEP 3: Assign data to the new node
STEP 4: Set temp to START and prev to temp
STEP 5: IF temp==NULL goto STEP 9
STEP 6: IF temp->data > data of new node, Goto STEP 8
STEP 7: Move temp to next node: temp=temp->next and prev=temp. Goto STEP 5
STEP 8: Set: prev>next to newnode and newnode>next to temp.
STEP 9: EXIT
*/

void insert_sorted_list()
{
    int ele;
    node *newnode;
    node *prev,*temp;
    printf("Enter the element\n");
    scanf("%d",&ele);
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    prev=start;
    temp=start;
    while(temp!=NULL)
    {
        if(temp->data>ele)
        {
            break;
        }
        else
        {
            prev=temp;
            temp=temp->next;
        }
    }
    prev->next=newnode;
    newnode->next=temp;


}

void traverse()
{
   node *temp=start;
   while(temp!=NULL)
   {
       printf("%d->",temp->data);
       temp=temp->next;
   }
}

void main()
{

insert_begining();
    insert_begining();
    traverse();
    insert_after();
    traverse();
    insert_sorted_list();
    traverse();

}
```

Deletion of an existing node from the linked list:

Existing nodes from a singly linked list (SLL) can be deleted from any position. Specifically following 2 cases are discussed:

(i) Deletion of node following a given node
(ii) Deletion of node matching item in the linked list

For first case, location is given after which node has to be deleted. For the second case, node matching item is found and deleted. If such item does not exist, no nodes are deleted.

Case (i): C program for Deletion of a node following given node:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{

    int data;
    struct node *next;
}node;
node *start=NULL;
void insert_node(int ele)
{
    node* newnode;
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->next=start;
    start=newnode;
}

/**
Algorithm to delete a node after given location
  Step 1: Initialize count to 1
  Step 2: Initialize temp to start and declare cur as node pointer.
  Step 3: If temp is not NULL
       3.1.: Check count is same as user given node location (loc)
       3.2: If same Goto Step 4
       3.3: Move temp to temp->next
       3.4: Increment count
       3.5: Goto Step 3
Step 4: Assign cur as temp->next
Step 5: Assign temp->next to cur->next
Step 6: Free cur node
Step 7: Exit
```

```
**/

void delete_node_after(int loc)
{
   int cnt=1;
   node* temp=start,*cur;
   while(temp!=NULL)
   {
      if(cnt==loc)
         break;
      temp=temp->next;
      cnt++;
   }
   cur=temp->next;
   temp->next=cur->next;
   free(cur);

}
void display()
{
   node*temp=start;
   while(temp!=NULL)
   {
      printf("%d->",temp->data);
      temp=temp->next;
   }
}

void main()
{
   insert_node(20);
   insert_node(30);
   insert_node(21);
   display();

  delete_node_after(1);
  display();

}
```

Case (ii): C program for Deletion of a node with matching item:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{

   int data;
   struct node *next;
}node;
```

```
node *start=NULL;
void insert_node(int ele)
{
   node* newnode;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=start;
   start=newnode;
}

/**
Algorithm to delete a node with matching item
  Step 1: Initialize temp and prev node pointers to start.
  Step 2: If temp is not NULL
       2.1.: Check temp->data and item
       2.2: If same Goto Step 3
       2.3: Assign prev as temp
       2.4: Assign temp as temp->next
       2.5: Goto Step 2
Step 3: Assign prev->next as temp->next
Step 4: Free temp node
Step 5: Exit
**/

void delete_match(int item)
{
    node*temp=start,*prev=start;
   while(temp!=NULL)
   {

      if(temp->data==item)
         break;
       prev=temp;
      temp=temp->next;
   }
   prev->next=temp->next;
   free(temp);
}

void display()
{
   node*temp=start;
   while(temp!=NULL)
   {
      printf("%d->",temp->data);
      temp=temp->next;
   }
}

void main()
```

```
{
    insert_node(20);
    insert_node(30);
    insert_node(21);
   display();
   delete_match(30);
    display();
}
```

Sorting Linked Lists:
Linked lists sorting involves arranging items inside the linked list is an order. The order can be ascending or descending. A variation of bubble sort algorithm is presented, in which there are nested loops. In outer loop, each element is passed from linked list and in inner loop its compared with all remaining elements and on out of order, swapping is done. It is an inplace sorting, as existing linked list is sorted and no new memory is required.

Algorithm for sorting linked lists:

```
STEP 1: REFER LIST BY A POINTER T
STEP 2: LET S=T->NEXT
STEP 3: IF S->DATA < T->DATA, SWAP S->DATA AND T->DATA
STEP 4 MOVE S to S->NEXT
STEP 5: IF S->NEXT==NULL,
                MOVE T to T->NEXT AND GOTO STEP 2
STEP 6: ELSE GOTO STEP 3
STEP 7: EXIT
```

C program for sorting of a linked list:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{

    int data;
    struct node *next;
}node;
node *start=NULL;
void insert_node(int ele)
{
    node* newnode;
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->next=start;
    start=newnode;
}

void sort_list()
{
    int temp;
     node*t=start,*s;
    while(t!=NULL)
    {
        s=t->next;
        while(s!=NULL)
        {
            if(s->data<t->data)
            {
                temp=s->data;
                s->data=t->data;
                t->data=temp;
            }
            s=s->next;
        }
        t=t->next;
    }
}

void main()
{
    insert_node(20);
    insert_node(30);
    insert_node(21);
    display();
    sort_list();
    display();

}
```

Merging Linked Lists:

Merging is an operation to copy the nodes of two linked lists into a single linked list. There are 2 variations of such merging:

- **(i) Concatenation – Copy data from list1 followed by data from list 2**
- **(ii) Sorted Merge – During copy make sure the lowest is copied and resulting output list is sorted.**

Algorithm-SIMPLE MERGE

```
STEP 1: REFER LIST1 BY A POINTER TEMP1
STEP 2: REFER LIST2 BY A POINTER TEMP2
STEP 3: IF TEMP1!=NULL
            3.1 ADD TEMP1 to END OF MERGEDLIST
            3.2 MOVE TEMP1=TEMP1->NEXT
            3.3 GOTO STEP3
STEP 4: IF TEMP2!=NULL
            4.1 ADD TEMP2 to END OF MERGEDLIST
            4.2 MOVE TEMP2=TEMP2->NEXT
            4.3 GOTO STEP4
STEP 5 EXIT
```

Corresponding C program:

```
#include<stdlib.h>
typedef struct
```

```
{
   int data;
   struct node *next;
}node;
node *start1=NULL,*start2=NULL,*start3=NULL;
node* insert_front(node*s,int ele)
{
   node* newnode;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=s;
   s=newnode;
   return s;
 }
 void display(node *s)
 {
    node*temp=s;
    while(temp!=NULL)
    {
       printf("%d->",temp->data);
       temp=temp->next;
    }
    printf("\b\b  ");
 }
 node* concat(node*start1, node*start2,
             node *start3)
 {
    node* temp=start1,*temp3;
    node *newnode;
    //Append list1 to outputlist
    while(temp!=NULL)
    {
       newnode=malloc(sizeof(node));
       newnode->data=temp->data;
       newnode->next=NULL;
       if(start3==NULL)
          start3=newnode;
       else
       {
          temp3=start3;
          while(temp3->next!=NULL)
             temp3=temp3->next;
          temp3->next=newnode;
       }
       temp=temp->next;
    }
    //Appending List 2 to output list
    temp=start2;
    while(temp!=NULL)
    {
       newnode=malloc(sizeof(node));
       newnode->data=temp->data;
       newnode->next=NULL;
       if(start3==NULL)
          start3=newnode;
       else
       {
          temp3=start3;
          while(temp3->next!=NULL)
             temp3=temp3->next;
          temp3->next=newnode;
       }
        temp=temp->next;
    }
    return start3;
 }
 void main()
 {
    start1=insert_front(start1,22);
    start1=insert_front(start1,19);
    start1=insert_front(start1,15);
    display(start1);
    printf("\n");
    start2=insert_front(start2,23);
    start2=insert_front(start2,17);
    display(start2);
    printf("\n");
    start3=concat(start1,start2,start3);
    display(start3);
 }
```

Algorithm-SORTED MERGE

```
STEP 1: REFER LIST1 BY A POINTER TEMP1
STEP 2: REFER LIST2 BY A POINTER TEMP2
STEP 3: IF TEMP1!=NULL AND TEMP2!=NULL
        3.1: IF TEMP1->DATA<TEMP2->DATA
            3.1.1: ADD TEMP1 to MERGEDLIST,
                   TEMP1=TEMP1->NEXT, GOTO STEP 3
            3.1.2: ELSE ADD TEMP2 to MERGEDLIST.
                   TEMP2=TEMP2->NEXT, GOTO STEP 3
STEP 4: IF TEMP1!=NULL, COPY RESIDUAL NODES TO MERGEDLIST and EXIT
STEP 5 IF TEMP2!=NULL, COPY RESIDUAL NODES TO MERGEDLIST and EXIT
```

Corresponding C program:

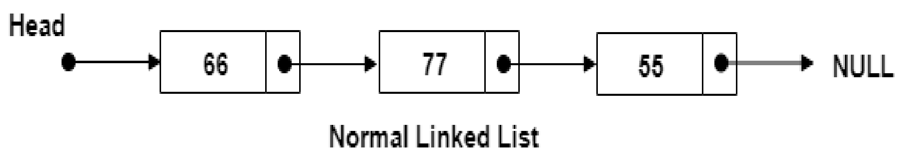
```
#include<stdlib.h>
typedef struct
{
   int data;
   struct node *next;
```

```
}node;

node* start1=NULL,*start2=NULL,*start3=NULL;
node* insert_front(int ele,node*start)
{
   node* newnode;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=start;
   start=newnode;
   return start;
}

void display(node *st)
{
   node* temp=st;
   while(temp!=NULL)
   {
      printf("%d->",temp->data);
      temp=temp->next;
   }
}

void insert_end(int ele)
{
   node*temp=start3;
   node*newnode;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=NULL;
   if(temp==NULL)
   {
      start3=newnode;
      return;
   }
   while(temp->next!=NULL)
      temp=temp->next;
   temp->next=newnode;
}

void sorted_merge()
{
   node*temp1=start1;
   node* temp2=start2;
   node* temp3=start3;
   while(temp1!=NULL && temp2!=NULL)
   {
      if(temp1->data < temp2->data)
      {
         insert_end(temp1->data);
```

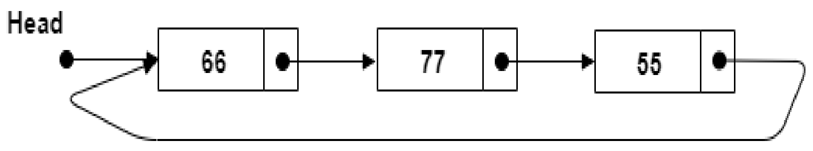
```
            temp1=temp1->next;
        }
        else
        {
            insert_end(temp2->data);
            temp2=temp2->next;
        }
    }
    while(temp1!=NULL) //copy residual nodes of list1
    {
        insert_end(temp1->data);
        temp1=temp1->next;
    }
    while(temp2!=NULL) //copy residual nodes of list2
    {
        insert_end(temp2->data);
        temp2=temp2->next;
    }
}


void main()
{

    start1=insert_front(22,start1);
    start1=insert_front(11,start1);
    start1=insert_front(10,start1);
    display(start1);
    printf("\n");
    start2=insert_front(16,start2);
    start2=insert_front(9,start2);

    display(start2);
    printf("\n");
    sorted_merge();
    display(start3);


}
```

Circular Linked List:

Circular Linked List is a variation of Singly Linked List where last NODE is connected back to first NODE instead of NULL. Eg:



Normal Linked List



Circular Linked List

Circular linked lists have a number of advantages over traditional linked lists.

- As there is no defined beginning or end to the list, data can be added and removed from the list at any time. This makes circular linked lists ideal for applications where data needs to be constantly added or removed, such as in a real-time application.
- Since data is stored in a ring-like structure, it can be accessed in a continuous loop. This makes circular linked lists ideal for applications where data needs to be processed in a continuous loop, such as in a real-time application or simulation.
- As there is no defined beginning or end to the list, circular linked lists are typically more efficient than traditional linked lists when it comes to memory usage. This is because traditional linked lists often require additional memory for pointers that point to the beginning and end of the list. Circular linked lists, on the other hand, only require a single pointer to be stored in memory – the head pointer.
- Circular linked lists are often easier to implement than traditional linked lists. This is because traditional linked lists often require the use of additional data structures, such as stacks and queues, to keep track of the list's beginning and end. Circular linked lists, on the other hand, only require a singly linked list data structure.

Implementation of various operations of circular linked list:

```
#include<stdlib.h>
typedef struct
{
   int data;
   struct node *next;
} node;

node *start=NULL;
```

//Insert a NODE to the beginning of Circular Linked List

```
void insert_begining(int ele)
{
   node* newnode,*temp;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
```

```
    if(start==NULL) //Nothing is der
    {
        start=newnode;
        newnode->next=start;
         return;
    }
    else //Something is der
    {
        temp=start;
        while(temp->next!=start)
        {
            temp=temp->next;
        }
        temp->next=newnode;
        newnode->next=start;
        start=newnode;

    }
}
```

//Insert a NODE to the end of Circular Linked List

```
void insert_end(int ele)
{
    node* temp=start;
     node* newnode;
    newnode=malloc(sizeof(node));
    newnode->data=ele;

    if(start==NULL)
    {
        start=newnode;
        start->next=start;
        return;
    }
    while(temp->next!=start)
    {
        temp=temp->next;
    }
    temp->next=newnode;
    newnode->next=start;
}
```

//Insert a NODE to a specific position of the Circular Linked List

```
void insert_specific_position(int ele,int loc)
{
    int cnt=1;
    node* temp=start;
    node* newnode;
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->next=NULL;
    while(temp->next!=NULL)
    {
        if(cnt==loc)
            break;
        cnt++;
        temp=temp->next;

    }
    newnode->next=temp->next;
    temp->next=newnode;
}
```

//Deleting a NODE at the beginning of Circular Linked List

```
void delete_front()
{
    node*cur=start,*temp=start;
    while(temp->next!=start)
        temp=temp->next;
    start=start->next;
    temp->next=start;
    free(cur);
}
```

//Deleting a NODE at the end of Circular Linked List

```
void delete_end()
{
    node*prev=start,*temp=start;
    while(temp->next!=start)
    {
        prev=temp;
          temp=temp->next;
    }
    prev-> next=start;
    free(temp);

}
```

//Deleting a NODE at the specific position of Circular Linked List

```
void delete_specific_pos(int loc)
{
     node*temp=start,*prev=start;
     int cnt=1;
    while(temp->next!=start)
    {
         if(cnt==loc)
            break;
         cnt++;
         prev=temp;
          temp=temp->next;
    }
    prev->next=temp->next;
    free(temp);
}
```

//Displaying (Traversing) a Circular Linked List

```
void display()
{
    node*temp=start;
    while(temp->next!=start)
    {
        printf("%d->",temp->data);
        temp=temp->next;
    }
    printf("%d->",temp->data);
}
```

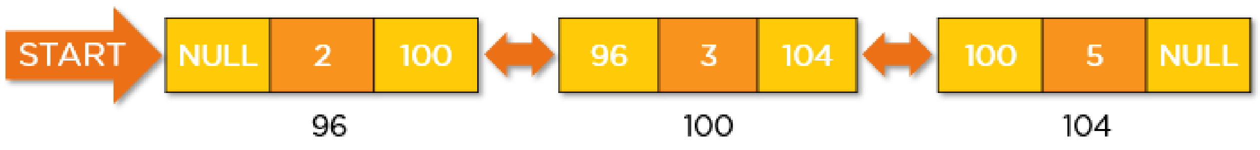
//Main driver program

```
void main()
{
    insert_begining(20);
    insert_begining(30);
    insert_end(40);
    insert_specific_position(19,2);
    display();
    printf("\n");
  //  delete_end();
  //  delete_front();
   delete_specific_pos(2);
    display();
}
```

Doubly Linked List (DLL):

A doubly linked list is a bi-directional linked list. Hence one can traverse it in both directions. Unlike singly linked lists, its nodes contain one extra pointer called the previous pointer. This pointer points to the previous node.

Eg:



It is advantageous over other data structures because it allows for quick insertion and deletion of elements. Additionally, it is easy to implement and can be used in a variety of applications.

Implementation of various operations of DLL:

```
#include<stdlib.h>
typedef struct
{
    int data;
    struct node*prev;
    struct node*next;
}node;

node*start=NULL;
```

//Inserting a node at front of DLL

```
void insert_front(int ele)
{
    node*newnode;
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->prev=NULL;
    newnode->next=NULL;
    if(start==NULL)
        start=newnode;
    else
    {

        newnode->next=start;
        start->prev=newnode;
        start=newnode;
    }
}
```

//Inserting a node at end of DLL

```
void insert_end(int ele)
{
     node*newnode,*temp=start;
    newnode=malloc(sizeof(node));
    newnode->data=ele;
    newnode->prev=NULL;
    newnode->next=NULL;
    if(start==NULL)
        start=newnode;
    else
    {
       while(temp->next!=NULL)
           temp=temp->next;
       temp->next=newnode;
       newnode->prev=temp;

    }
}
```

//Inserting a node after the specified position in the DLL
```
void insert_specific_position(int pos,int ele)
{
   node*newnode,*temp=start,*ntemp;
   int cnt=1;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->prev=NULL;
   newnode->next=NULL;

      while(temp->next!=NULL)
      {

         if(cnt==pos)
            break;
         cnt++;
         temp=temp->next;
      }
        ntemp=temp->next;
      newnode->next=ntemp;
      temp->next=newnode;
      newnode->prev=ntemp;
      ntemp->prev=newnode;

   }
```

//Deleting a node at front of DLL
```
void delete_front()
{

   node *temp=start;
   start=start->next;
   start->prev=NULL;
   free(temp);
}
```

//Deleting a node at end of DLL
```
void delete_rear()
{
   node* temp=start,*ntemp;
   while(temp->next!=NULL)
      temp=temp->next;
   ntemp=temp->prev;
   ntemp->next=NULL;
   free(temp);
}
```

//Deleting a node in the specified position of DLL
```
void delete_specific_pos(int pos)
{
    node*temp=start,*ntemp,*ftemp;
    int cnt=1;
    while(temp->next!=NULL)
    {
        if(cnt==pos)
            break;
        cnt++;
        temp=temp->next;
    }
    ntemp=temp->prev;
    ntemp->next=temp->next;
    ftemp=temp->next;
    ftemp->prev=ntemp;
    free(temp);
}
```

//Displaying a DLL

```
void display()
{
    node *temp=start;
    while(temp!=NULL)
    {
        printf("%d->",temp->data);
        temp=temp->next;
    }
}
```

//Reversing a DLL

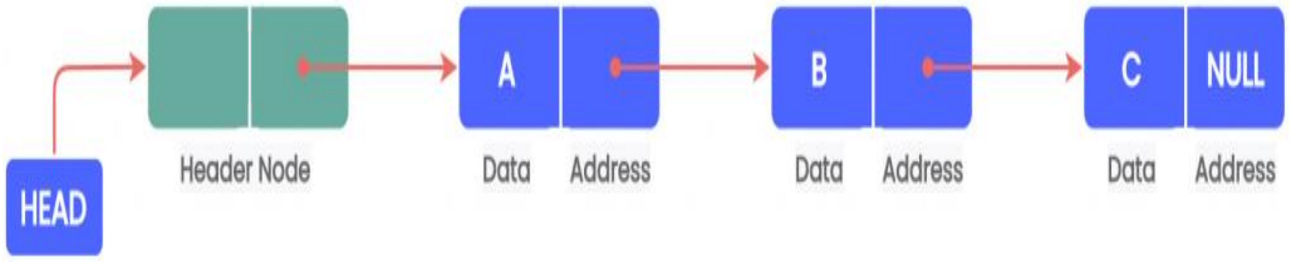
```
void reverse_list()
{

    node *temp=start;
    while(temp->next!=NULL)
        temp=temp->next;
    while(temp!=NULL)
    {
        printf("%d<-",temp->data);
        temp=temp->prev;
    }
}
```

//Main driver for various operations of DLL

```
void main()
{
    insert_front(20);
    insert_front(30);
    insert_end(50);
    insert_end(11);
    insert_specific_position(2,33);
    delete_specific_pos(2);
    display();
    printf("\n");
    reverse_list();
}
```

Header Linked Lists (HLL)

Linked lists in which starting node is a special node called Header node. Header node does not have any data. It is used to connect to the other nodes. Eg:



The header node does not represent an item in the linked list. This data part of this node is generally used to hold any global information about the entire linked list. The next part of the header node points to the first node in the list.

A header linked list can be divided into two types:

- **Grounded header linked list** that stores NULL in the last node's next field.
- **Circular header linked list** that stores the address of the header node in the next part of the last node of the list.

Advantages of Header Linked lists:

(i) Coding of insertion and deletion becomes easier as one need not worry about nothing or something is there in the linked list.

(ii) Further header node data can be used to record number of nodes in the linked list. Hence to know the number of nodes, one need not traverse linked list again and again.

Implementation of various operations of HLL:

```
#include<stdlib.h>
typedef struct
{
    int data;
    struct node*next;
}node;
node *head;
```

//Creating Header Node

```
void create_header_node()
{
    head=malloc(sizeof(node));
    head->next=NULL;
}
```

```
//Inserting a node to front of HLL
void insert_front(int ele)
{
   node* newnode;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=head->next;
   head->next=newnode;
}

//Inserting a node to end of HLL
void insert_end(int ele)
{
   node* newnode,*temp=head;
   newnode=malloc(sizeof(node));
   newnode->data=ele;
   newnode->next=NULL;
   while(temp->next!=NULL)
      temp=temp->next;
   temp->next=newnode;
}

//Displaying all the nodes of HLL (Traversal)
void display()
{
   node*temp=head->next;
   while(temp!=NULL)
   {
      printf("%d->",temp->data);
      temp=temp->next;
   }
}

//Deleting a node at the front of HLL
void delete_front()
{
   node*temp=head->next;
   head->next=temp->next;
   free(temp);
}

//Deleting a node at the end of HLL
void delete_end()
{
   node*temp=head,*prev=head;
   while(temp->next!=NULL)
   {
      prev=temp;
      temp=temp->next;
   }
```

```
    prev->next=NULL;
    free(temp);
}
```

//Main program driver to demonstrate various operations of HLL

```
void main()
{
    create_header_node();
    insert_front(40);
    insert_front(22);
    insert_end(23);
    insert_end(10);
    delete_front();
    delete_end();
    display();
}
```

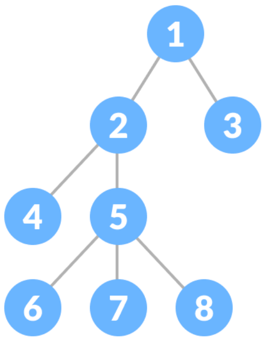
Module-4

Trees

Overview of a Tree:

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges. Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. This behaviour is not acceptable in today's computational world. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

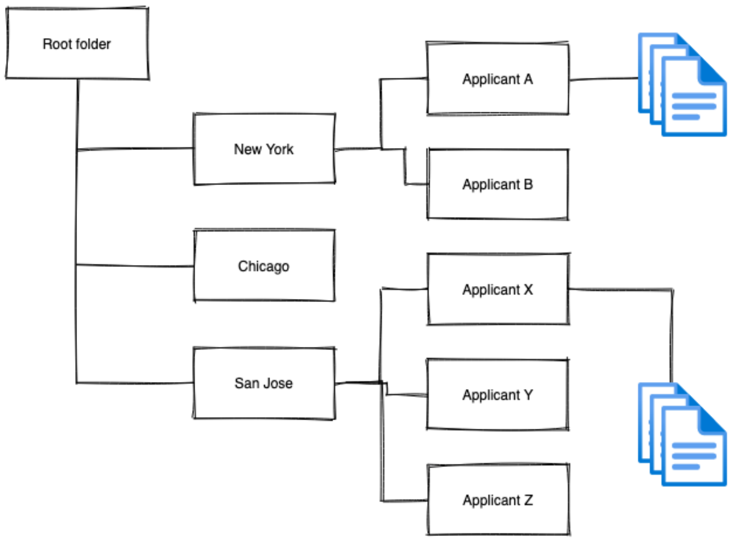
Eg: A Tree with 8 nodes having 4 levels. Different nodes connected with parent-child relationship. These trees grow top down and special node at the top is called root.



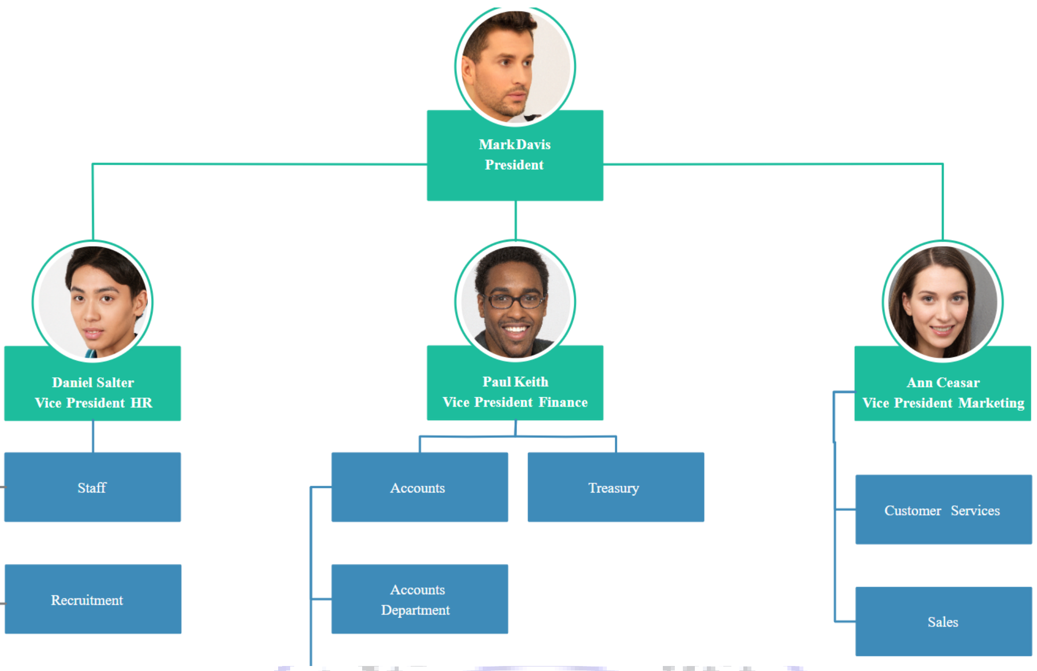
Real time Trees:

Various applications of trees do exist in real time and are as follows:

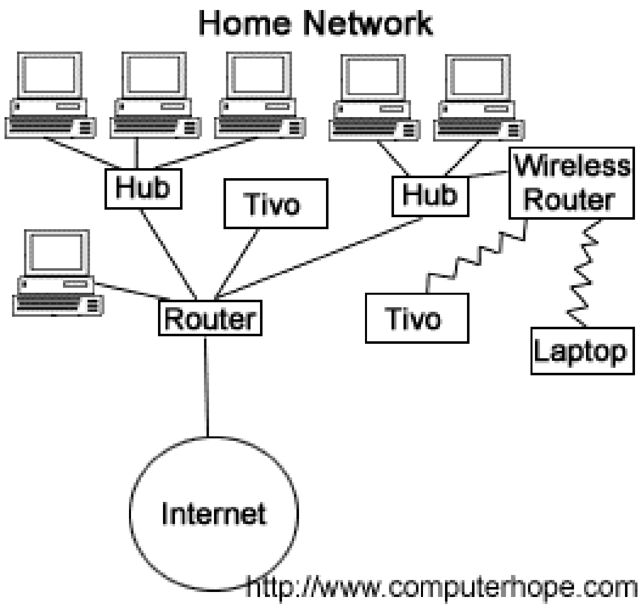
(i) Folders-Files hierarchy in Operating system and cloud:



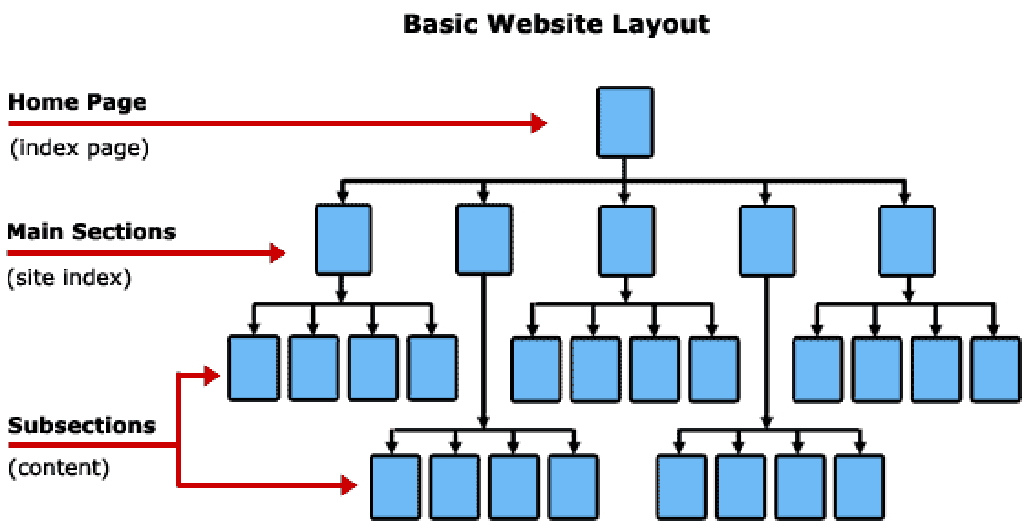
(ii) Every company/institution has an organization chart that shows the flow of authority in the organization. Such an organization chart is depicted below:



(iii) **Computers/ computing devices are interconnected to form a computer network as follows:**



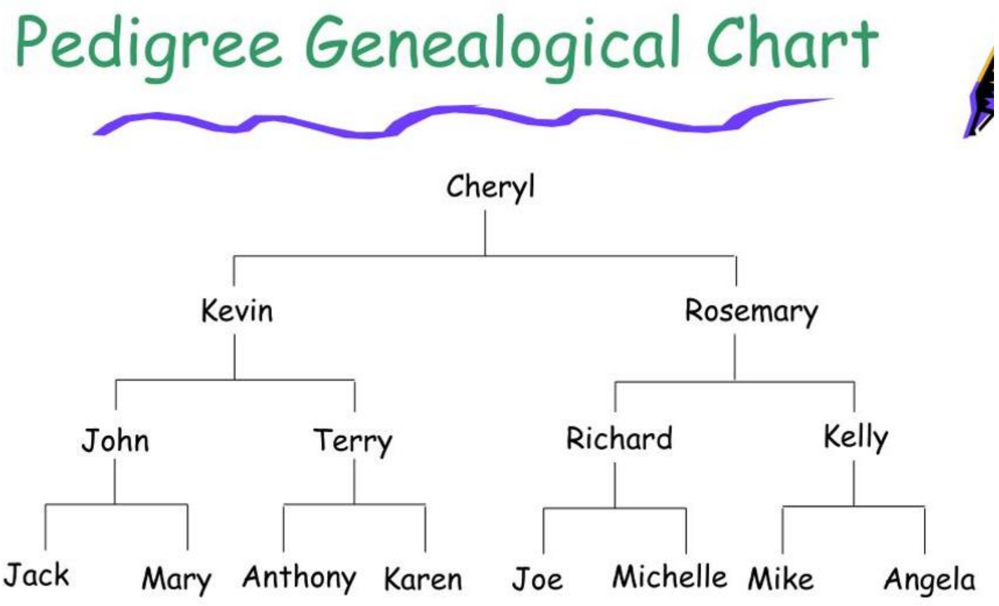
(iv) **The website for any firm/organization/industry will have various webpages organized as a tree fashion shown next:**



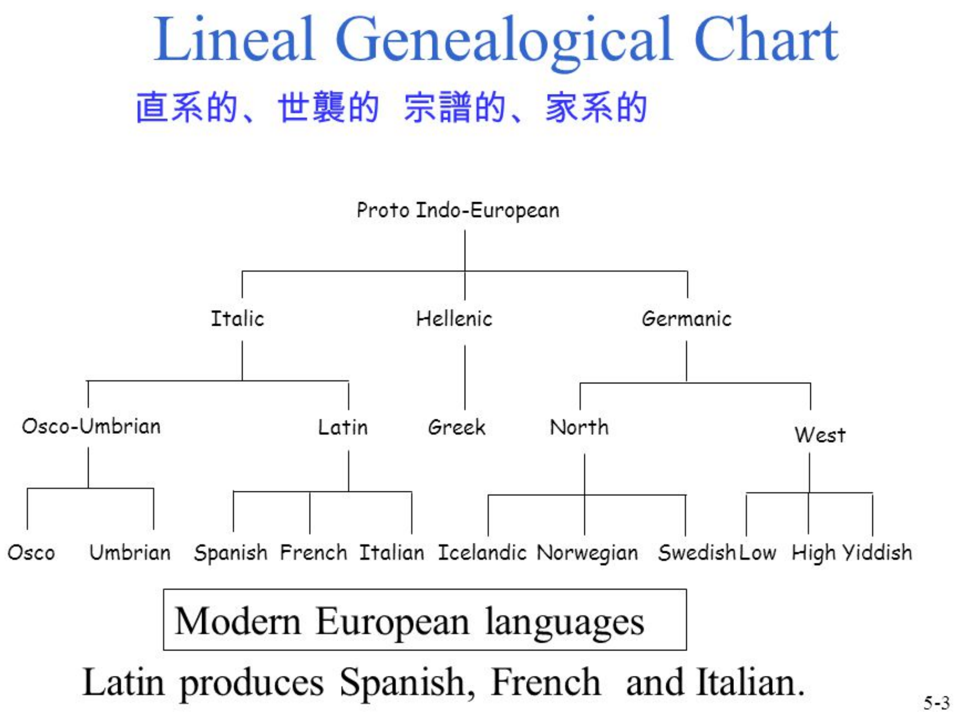
(v) Most trending network of the present era is the social network which is modelled as a tree with subscribers as nodes and their relationship as edges.

Ancestor and Descendant Trees:
Ancestor Tree is a tree in which children of each node are its ancestors. Hence path from root to any node gives the list of ancestors. One such ancestor tree is as follows. In this example, ancestors of Kevin are: John, Terry (level1), Jack, Mary, Anthony, Karen (level2).

Descendant Tree is a tree in which children of each node are its descendants. Hence path from root to any node gives the list of descendants. One such descendant tree is as follows. In this example, descendants of Osco-Umbrian are: Osco and Umbrian.



Ancestor Tree



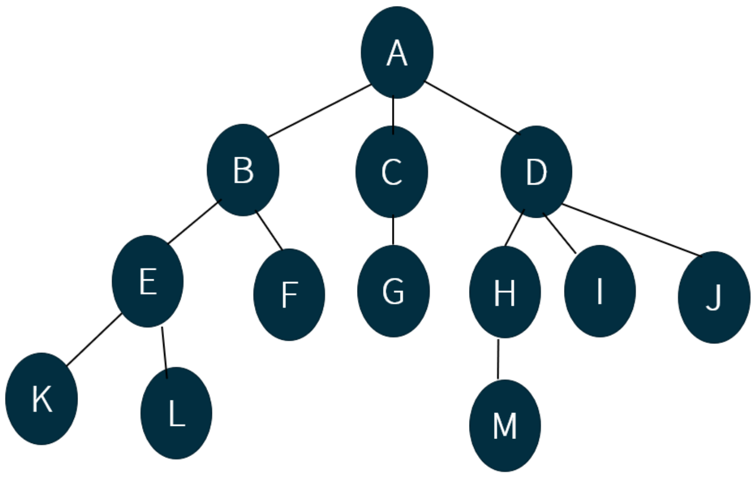
Descendant Tree

Definition of a tree:
A tree is a finite set of one or more nodes such that:

(1) *Specially designated node called root*
(2) *Remaining nodes are partitioned into n disjoint sets, each of which are subtrees*

Tree Terminologies:

Consider an example tree for stating various terminologies:



Node of a tree: Any item which as information and branch to other node. In case of leaf node, branch is NULL. Eg: in Node C, information is C and branch is to D.

Degree of a node: The number of subtrees of any node is its degree. Eg. B has degree 2 as it has 2 subtrees. If the degree of any node is 0, it is called Leaf Node. If degree of a node is non-zero, it is called Non-terminal node.

Siblings: Children of the same parent are called siblings. E and F are siblings as they are children of the same parent B.

Degree of a tree – max degree of any node is called its degree.

Ancestors: Ancestors of any node is the list of nodes in the path from root to that node.

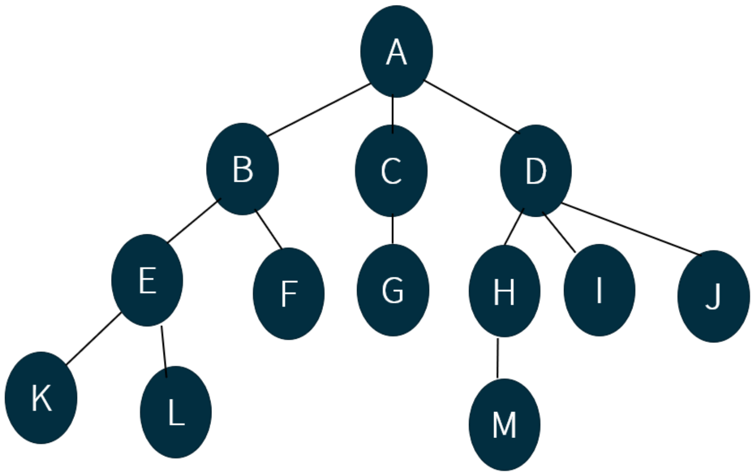
Level: Level of a tree starts with 1 at root node. Subsequently number increases with other descendants.

Height of a tree: Max level of a tree is the height of a tree.

Representation of Trees:
There are 3 ways of representing trees:
(i) List Representation
(ii) Leftchild, right sibling representation
(iii) Binary Tree (degree two) representation

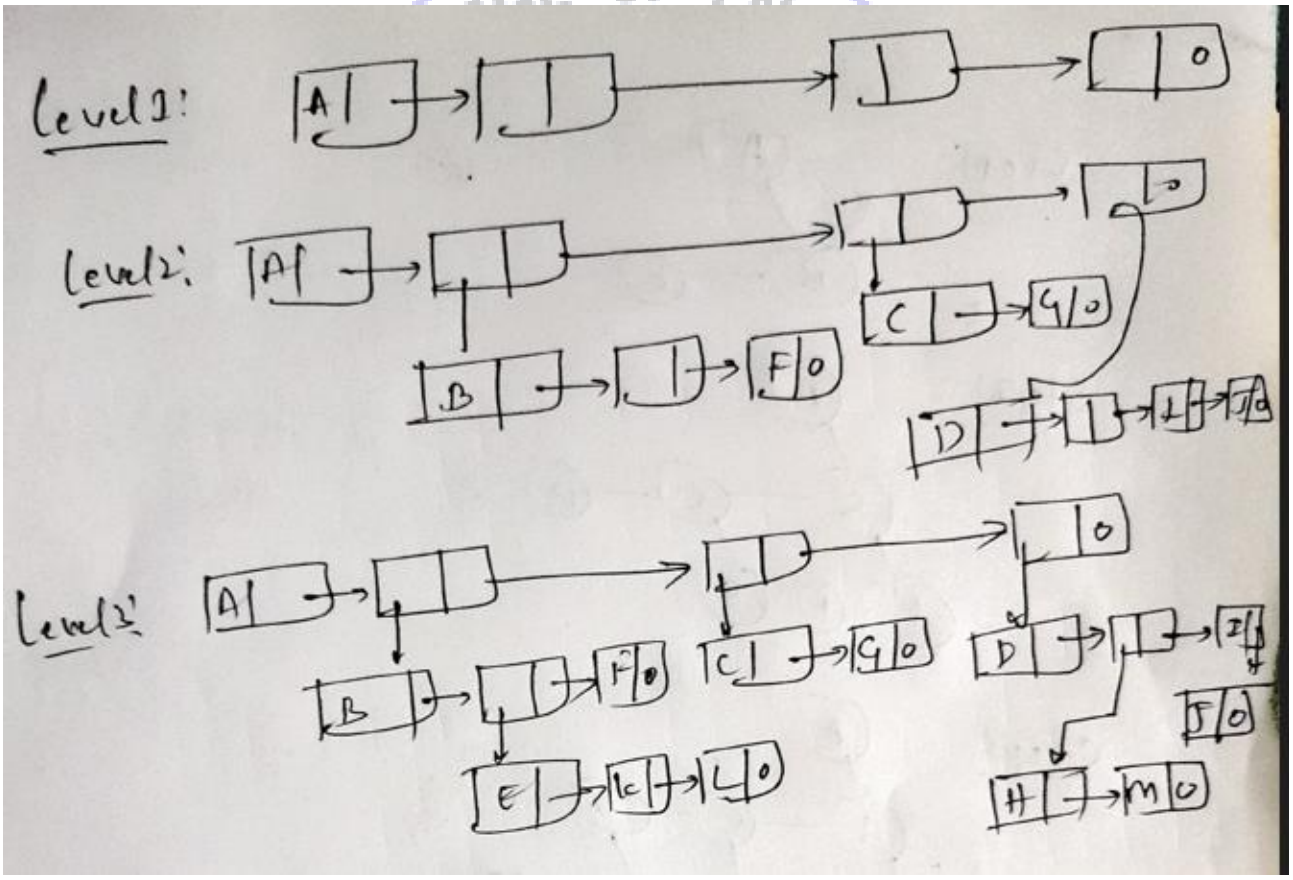
(i) List Representation:
In the list representation, level by level traversal is performed. For each node a linked list is created for its immediate children. For non-terminal nodes, data is left empty and pointer extended in next level. For leaf nodes., pointer portion of node in the linked list is made 0 (NULL).

Eg: For tree



:

Linked list representation proceeds as follows:

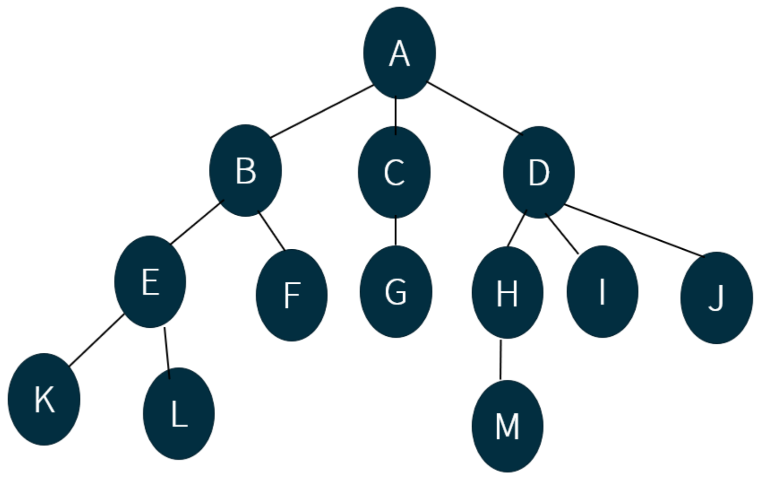


(ii) Left child -right sibling representation

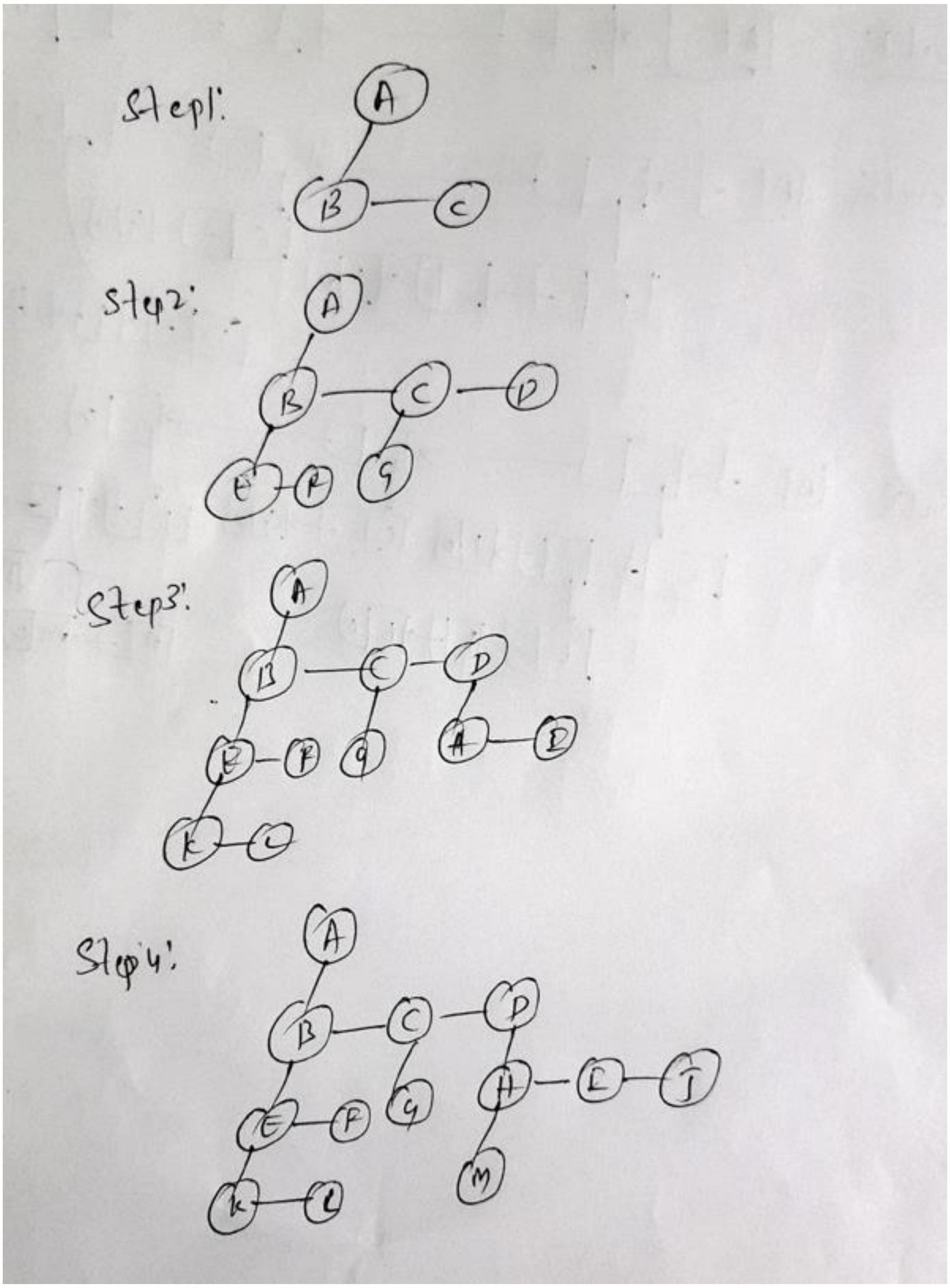
In left child-right sibling representation, any node will be converted to have only left child. The right child is reconnected as right sibling. The structure of any node in the tree is:

data	
Left child	Right sibling

Eg: For tree



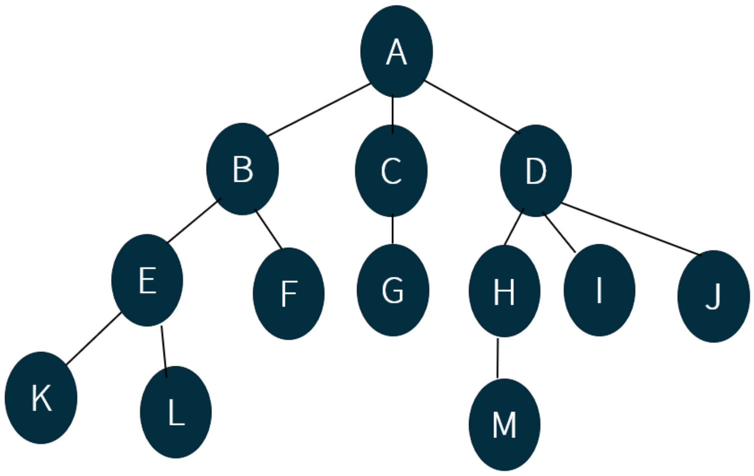
:
Left-child Right sibling representation proceeds as follows:



(ii) Binary Tree representation:

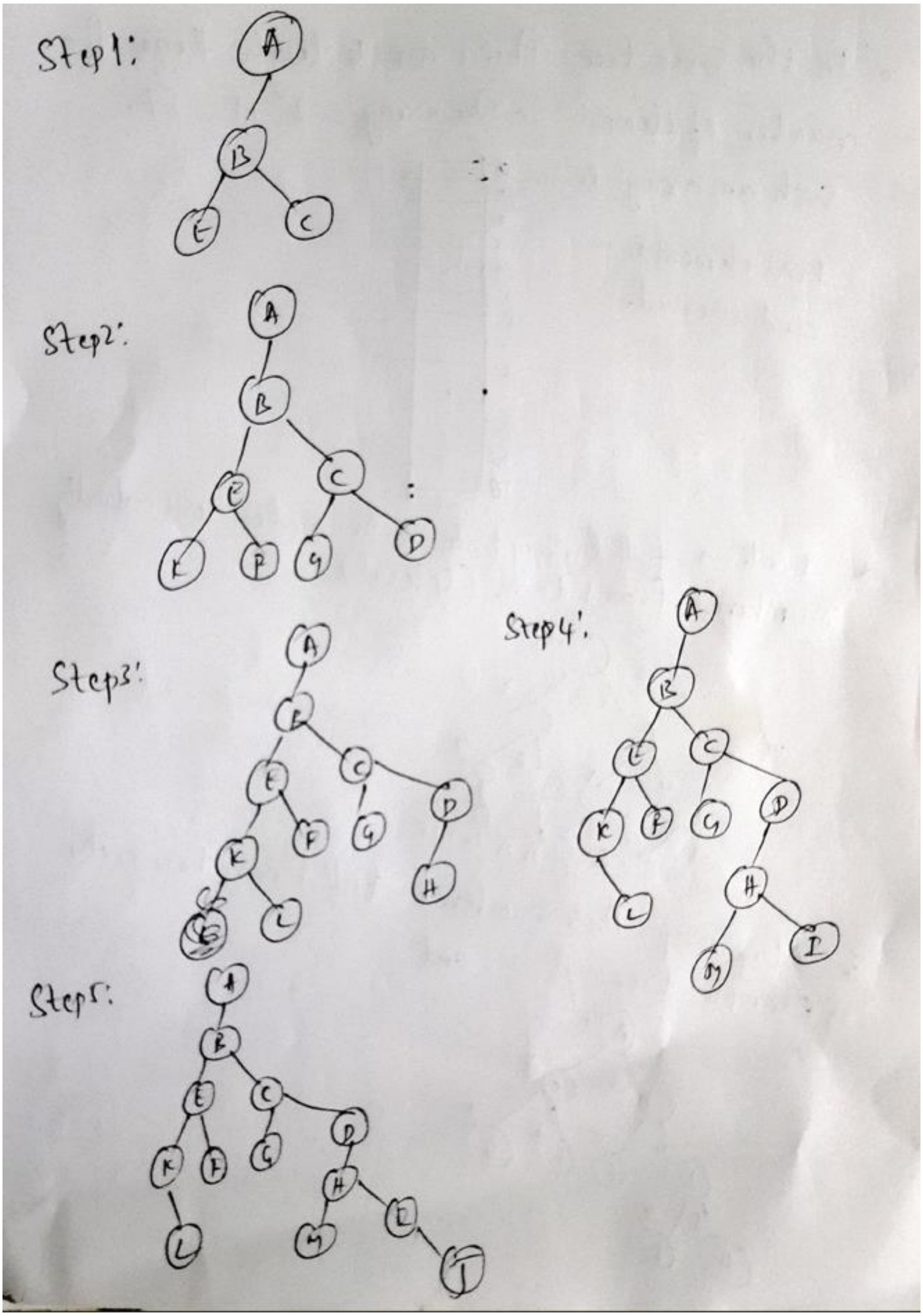
In the binary tree representation, each node can contain max two children. To get such a representation, tree is first converted to left-child, right sibling representation. Using such a representation, each right sibling is converted to right subtree.

Eg: For tree



:

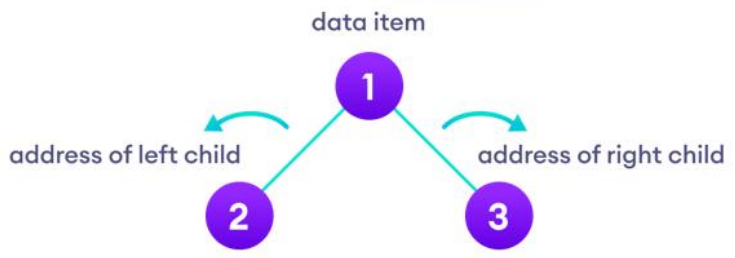
Binary Tree representation proceeds as follows:


Step1:
A
B
E
C
Step2:
A
B
E
C
K
F
G
D
Step3:
A
B
E
C
D
K
F
G
H
L
Step4:
A
B
E
C
K
F
G
D
L
H
M
I
Step5:
A
B
E
C
D
K
F
G
H
L
M
I
J

Binary Trees:

A binary tree is a finite set of nodes that is either empty or consists of root node and two disjoint binary trees called left subtree and right subtree. Each node of a binary tree consists of three items:

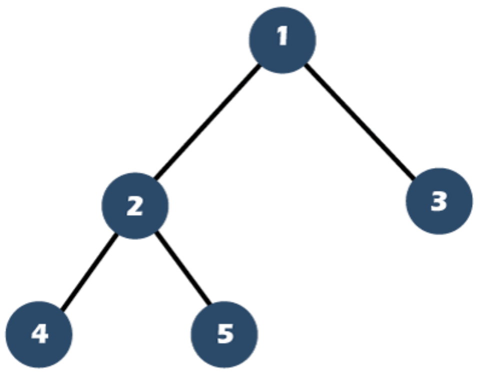
- data item
- address of left child
- address of right child



Types of Binary Tree

1. Full Binary Tree

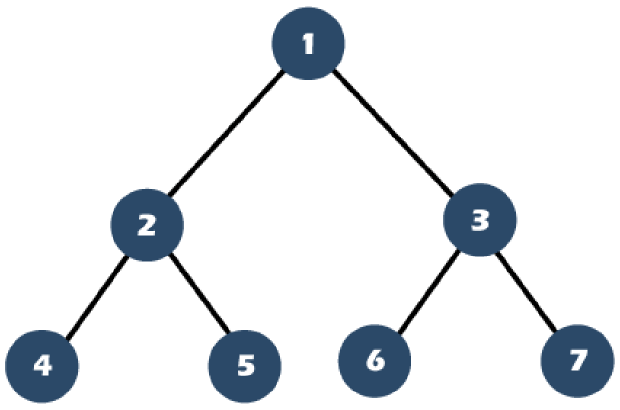
A full Binary tree is a special type of binary tree in which all the nodes have two children except the leaf nodes. Eg. full binary tree. This binary tree is a full binary tree as all the nodes except the leaf nodes have two children.



.

2. Complete Binary Tree:

A binary tree is said to be a complete binary tree when all the levels are completely filled except the last level, which is filled from the left. Eg. of complete binary tree – in this tree all levels are filled making it complete binary tree.



The complete binary tree is similar to the full binary tree except for the two differences which are given below:

- The filling of the leaf node must start from the leftmost side.
- It is not mandatory that the last leaf node must have the right sibling.

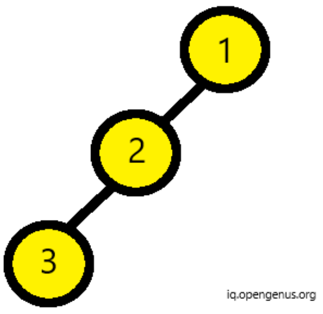
3. Skewed Trees:

A binary tree can be called a skewed binary tree if all nodes have one child or no child at all. They can be of two types:

- Left Skewed Binary Tree
- Right Skewed Binary Tree

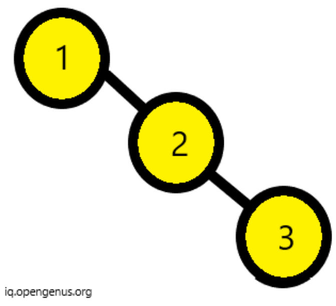
Left Skewed Binary Tree

If all nodes are having a left child or no child at all then it can be called a left skewed binary tree. In this tree all children at right side remain null.



Right Skewed Binary Tree

If all nodes are having a right child or no child at all then it can be called a right skewed binary tree. In this tree all children at left side remain null.



Abstract Data Type of a binary tree:

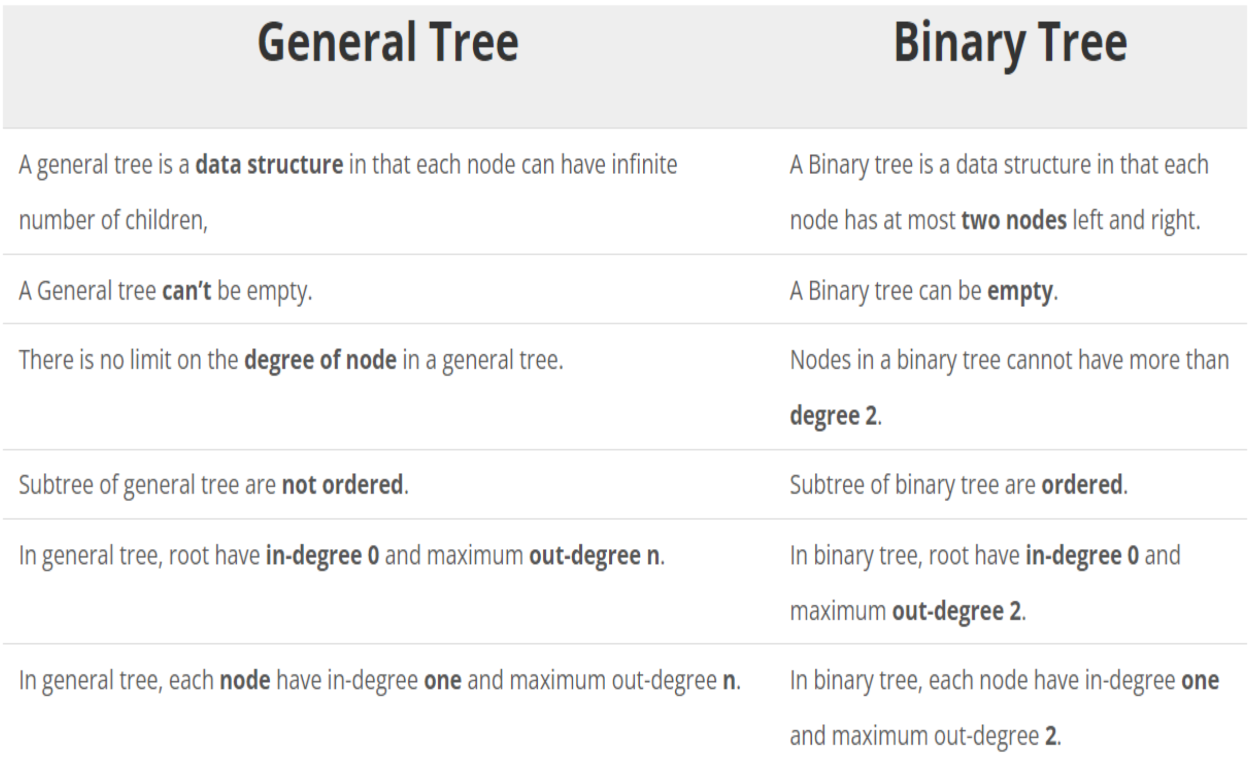
ADT Binary_Tree:

objects: finite set of nodes either empty or consisting of root node, left subtree and right subtree

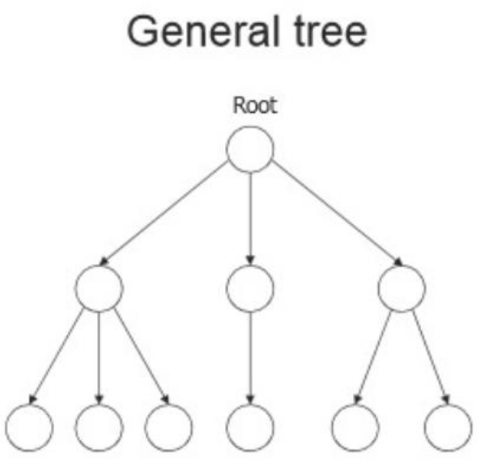
Functions:

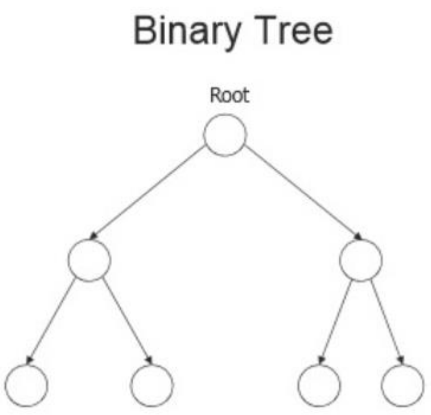
BinTree Create()	::=Creates an empty binary tree
Boolean IsEmpty(bt)	::=if(bt==empty binary tree) return TRUE else return FALSE
BinTree MakeBT(bt1, item, bt2)	::=return binary tree with root node item and left subtree bt1 and right subtree bt2
BinTree Lchild(bt)	::=if (IsEmpty(bt)) return error else return left subtree of bt
element Data(bt)	::=if (IsEmpty(bt)) return error else return data in the root node of bt
BinTree Rchild(bt)	::=if (IsEmpty(bt)) return error else return right subtree of bt

Trees v/s Binary Trees:



General Tree	Binary Tree
A general tree is a **data structure** in that each node can have infinite number of children,	A Binary tree is a data structure in that each node has at most **two nodes** left and right.
A General tree **can't** be empty.	A Binary tree can be **empty**.
There is no limit on the **degree of node** in a general tree.	Nodes in a binary tree cannot have more than **degree 2**.
Subtree of general tree are **not ordered**.	Subtree of binary tree are **ordered**.
In general tree, root have **in-degree 0** and maximum **out-degree n**.	In binary tree, root have **in-degree 0** and maximum **out-degree 2**.
In general tree, each **node** have in-degree **one** and maximum out-degree **n**.	In binary tree, each node have in-degree **one** and maximum out-degree **2**.





Binary Tree Representations:

There are 2 ways of representing binary trees:

i. Using Arrays
ii. Using Linked List

(i) Array Representation:

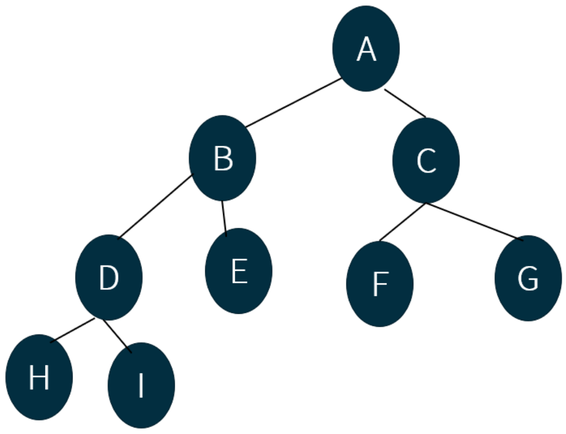
In this representation, an array is declared with size being (n+1) for an **n** node tree. First location of array (index 0) is empty. For remaining indices, elements will be filled such that following relationship is maintained for an node with index ‘i' in the array.

$$\boldsymbol{parent(i) = \lfloor i/2 \rfloor \; if \; i \neq 1, If \; i = 1, no \; parent}$$

$$\boldsymbol{leftChild(i) = 2i \; if \; 2i \leq n, Else \; no \; Left \; Child}$$

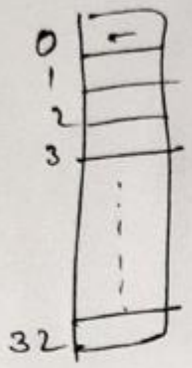
$$\boldsymbol{rightChild(i) = 2i + 1, if \; 2i + 1 \; \leq n, Else \; no \; Right \; Child}$$

For example tree:

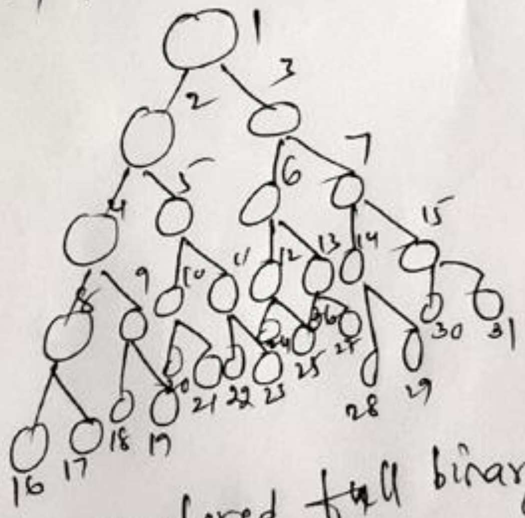


Array Representation is done as follows:

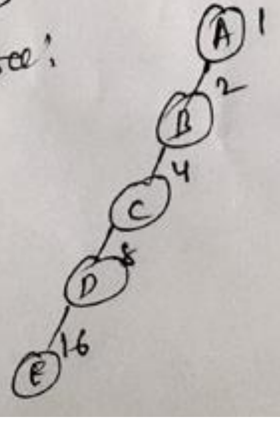
* In This tree, There are 5 levels. Hence The number of elements in The array $= 2^5 + 1 = 33$. Hence an array of 33 elements is created with index 0 reserved and tree elements stored from 1 to 32.

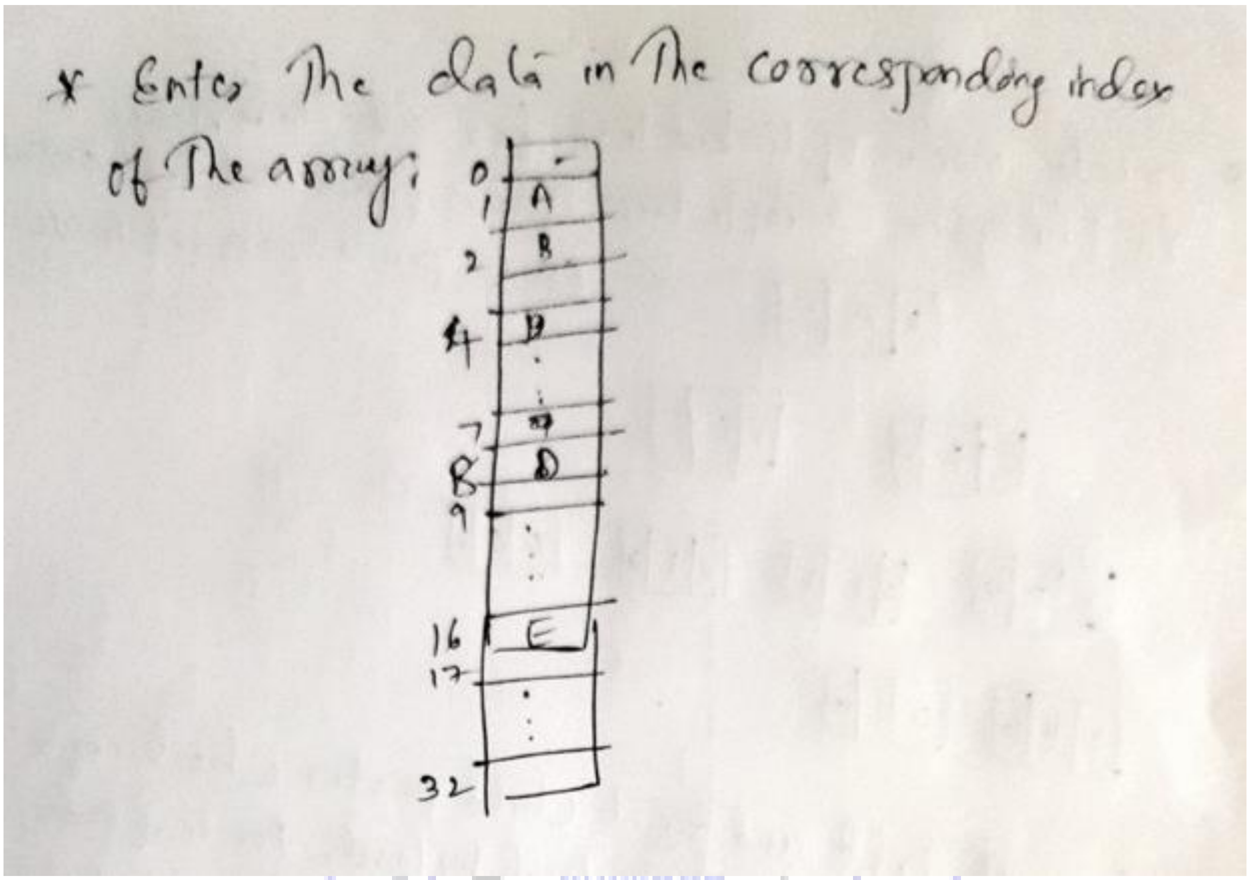


* Write a full binary tree with nodes sequentially numbered from 1 to 32. (5 levels $= 2^5$)



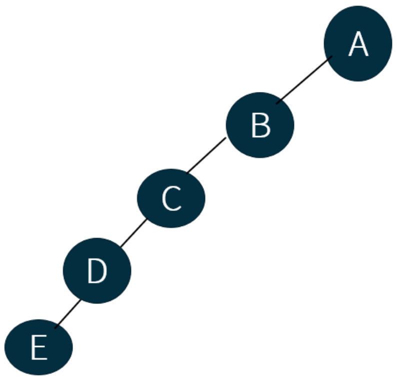
* Map This numbered full binary tree with example binary tree:



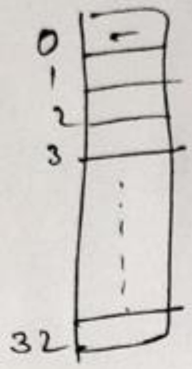


* Enter the data in the corresponding index of the array:

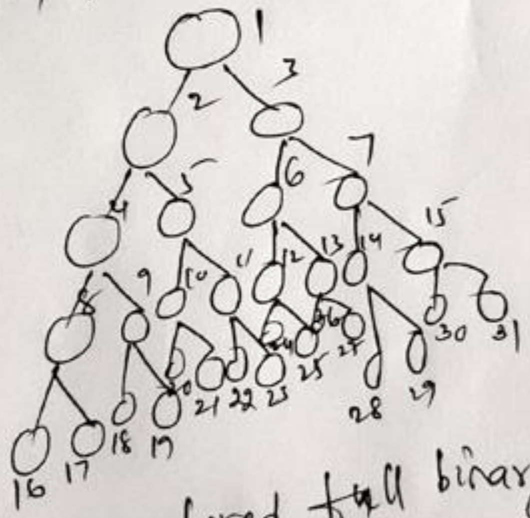
For example tree:



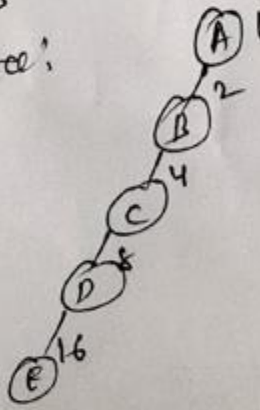
* In This tree, There are 5 levels. Hence The number of elements in The array $= 2^5 + 1 = 33$. Hence an array of 33 elements is created with index 0 reserved and tree elements stored from 1 to 32.

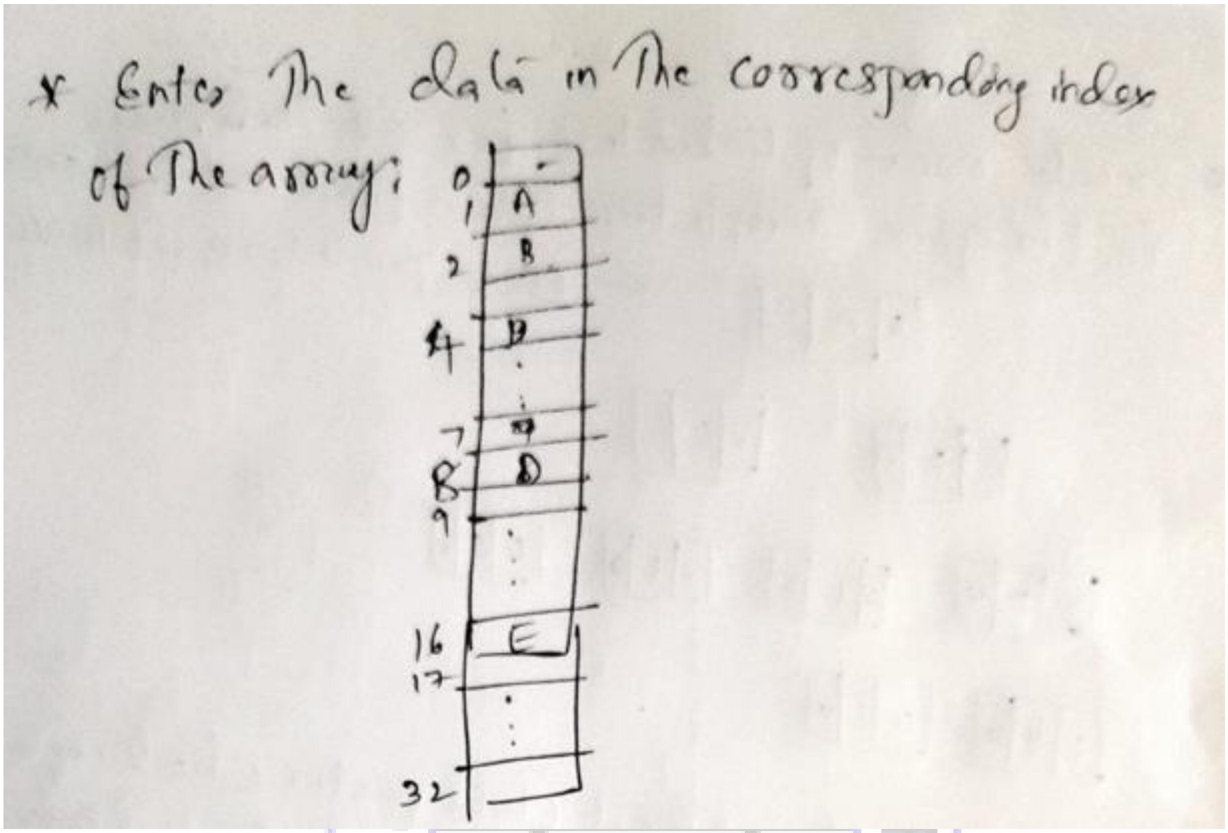


* Write a full binary tree with nodes sequentially numbered from 1 to 32. (5 levels = 2^5)



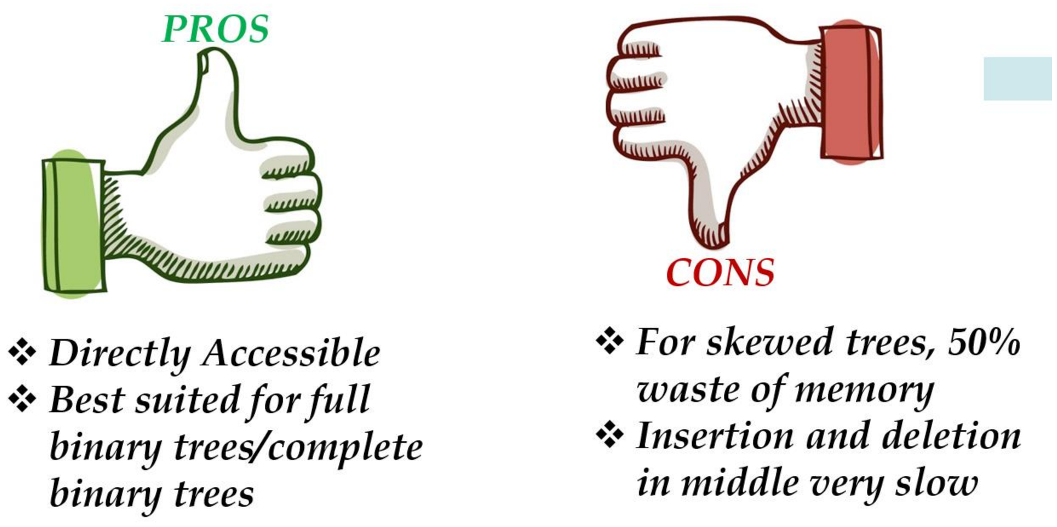
* Map This numbered full binary tree with example binary tree:




* Enter the data in the corresponding index of the array:

Array representation is done as follows:

Pros and cons of array representation:



- ***Directly Accessible***
- ***Best suited for full binary trees/complete binary trees***

- ***For skewed trees, 50% waste of memory***
- ***Insertion and deletion in middle very slow***

(ii) Linked List representation:
In the Linked List representation, for each node in the tree, a node in the linked list is created. Each node of linked list consists of following structure:

Node

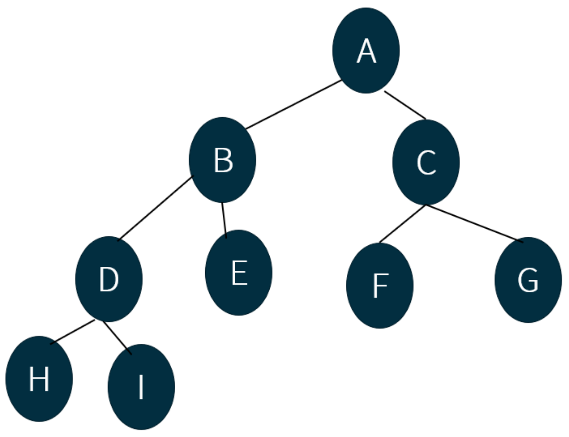
Left Child Pointer	Data	Right Child Pointer

C programming language struct for the node is as follows:

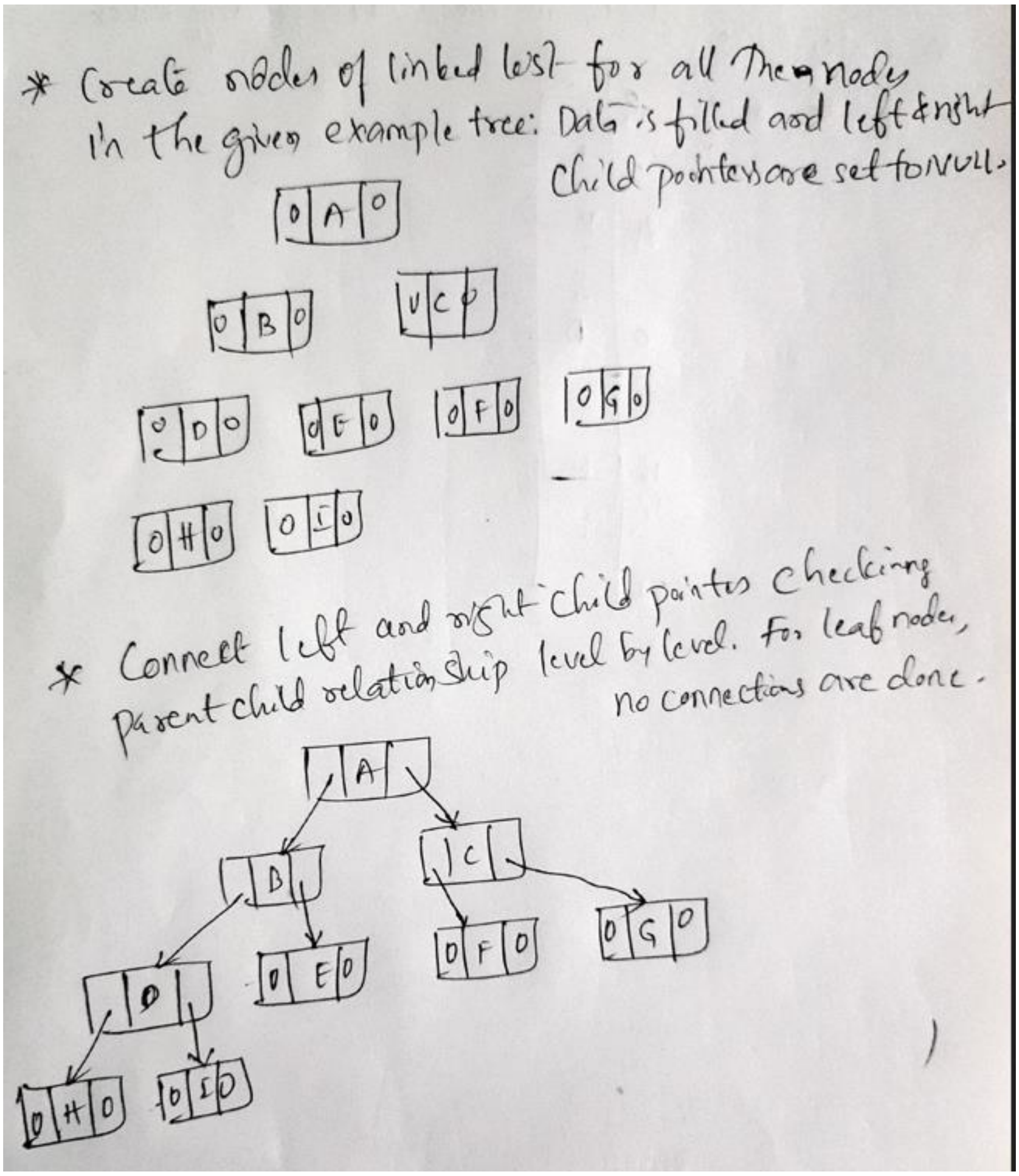
```
typedef struct
{
    int data;
    struct node *leftChild;
    struct node *rightChild;
}node;
```

Hence for all nodes in the tree, nodes in the linked list is created. Later left and right child connectivity through pointers are drawn.

For example tree:



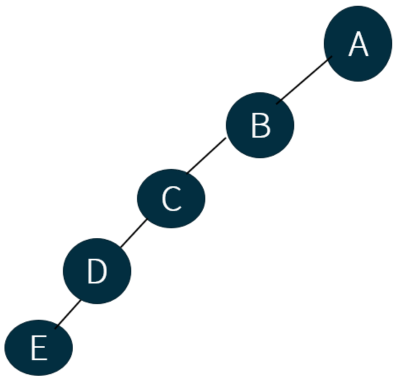
LinkedList Representation is done as follows:



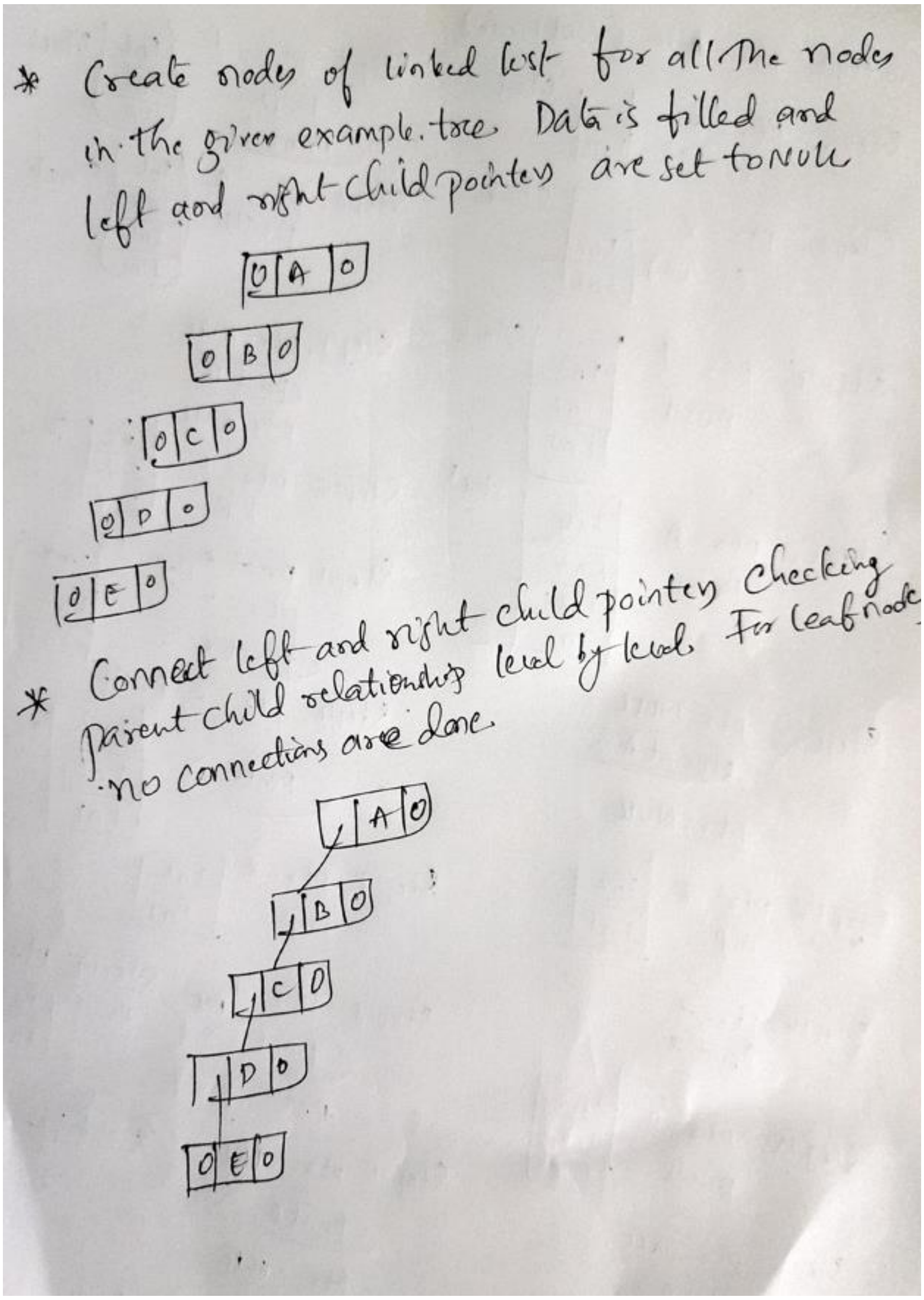
* Create nodes of linked list for all the nodes in the given example tree: Data is filled and left & right Child pointers are set to NULL.

* Connect left and right child pointers checking parent child relationship level by level. For leaf nodes, no connections are done.

For example tree:



LinkedList representation is done as follows:



* Create nodes of linked list for all the nodes in the given example tree. Data is filled and left and right child pointers are set to Null

* Connect left and right child pointers checking parent child relationship level by level. For leaf node no connections are done.

0 A 0
0 B 0
0 C 0
0 D 0
0 E 0
A 0
B 0
C 0
D 0
0 E 0

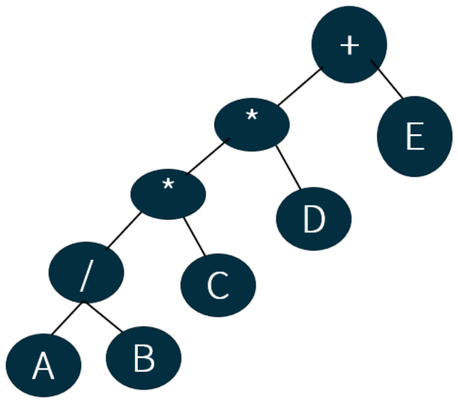
Binary Tree Traversal:
Traversal of a tree is a technique in which all nodes are visited. The order in which they are visited gives different traversal techniques:

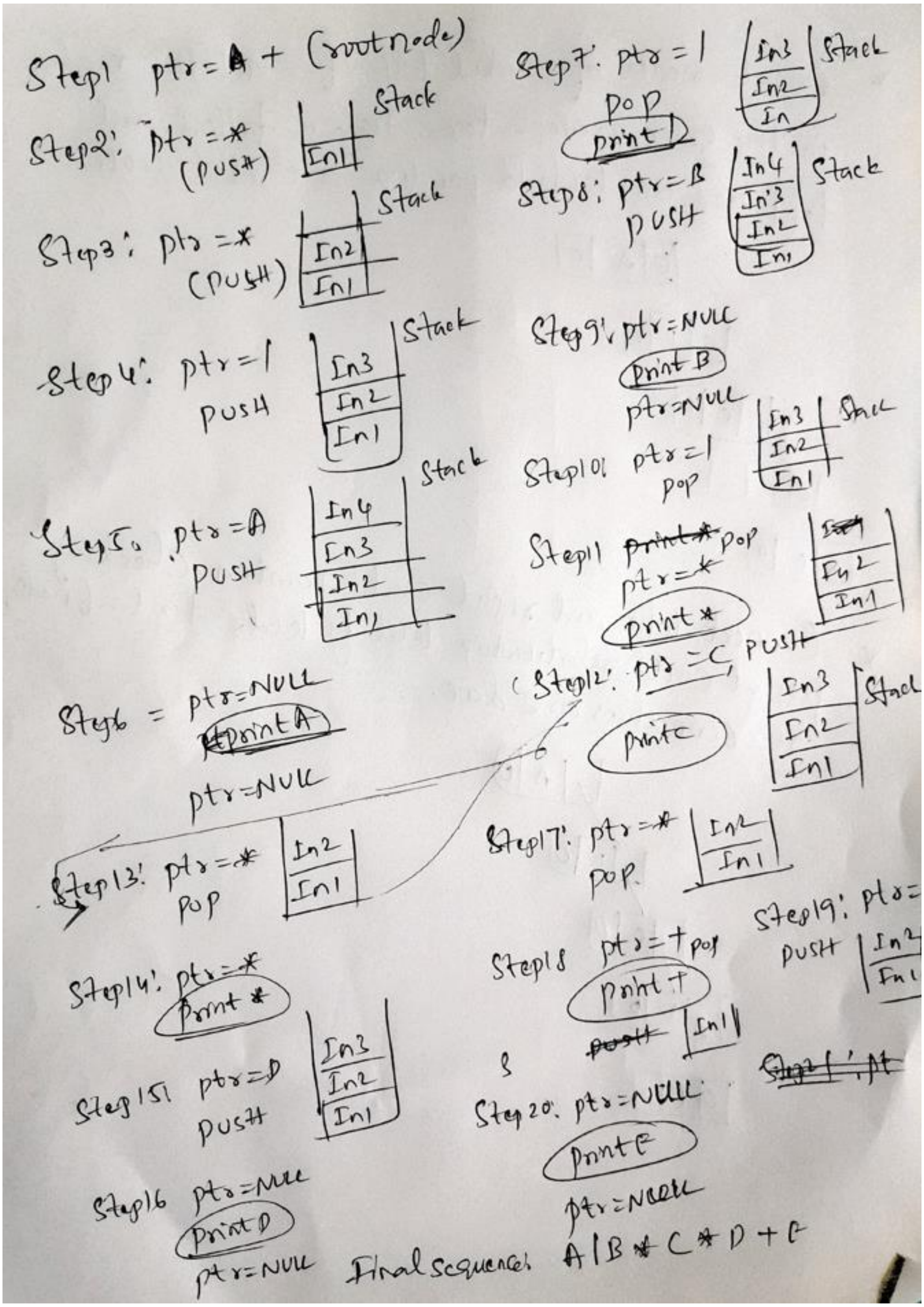
(i) Inorder:

In the Inorder traversal, node comes in between left and right child. Hence technique is to go to dead left and start printing nodes in LNR fashion for each subtree. The C recursive procedure for inorder traversal is:

```
void inorder(treePointer ptr)
{
     if(ptr)
    {
          inorder(ptr->leftChild);
          printf("%d",ptr->data);
          inorder(ptr->rightChild);
    }
}
```

Example inorder traversal for the tree:





Step1 ptr = ~~A~~ + (rootnode)

Step2: ptr = * (PUSH) — Stack: In1

Step3: ptr = * (PUSH) — Stack: In2, In1

Step4: ptr = / PUSH — Stack: In3, In2, In1

Step5: ptr = A PUSH — Stack: In4, In3, In2, In1

Step6 = ptr = NULL ~~print A~~ ptr = NULL

Step13: ptr = * pop — In2, In1

Step14: ptr = * Print *

Step15: ptr = D PUSH — In3, In2, In1

Step16 ptr = NULL Print D ptr = NULL

Step7: ptr = / pop print / — In3, In2, In — Stack

Step8: ptr = B PUSH — In4, In'3, In2, In1 — Stack

Step9: ptr = NULL Print B ptr = NULL — In3, In2, In1 — Stack

Step10: ptr = / pop

Step11 ~~print *~~ pop ptr = * print * — ~~In4~~, In2, In1

Step12: ptr = C, PUSH print C — In3, In2, In1 — Stack

Step17: ptr = * pop — In2, In1

Step18 ptr = + pop print + ~~push~~ — In1

Step19: ptr = PUSH — In2, In1

~~Step21: pt~~

Step20: ptr = NULL Print E ptr = NULL

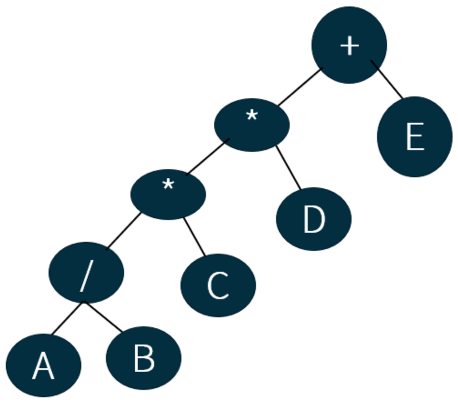
Final sequence: A / B * C * D + E

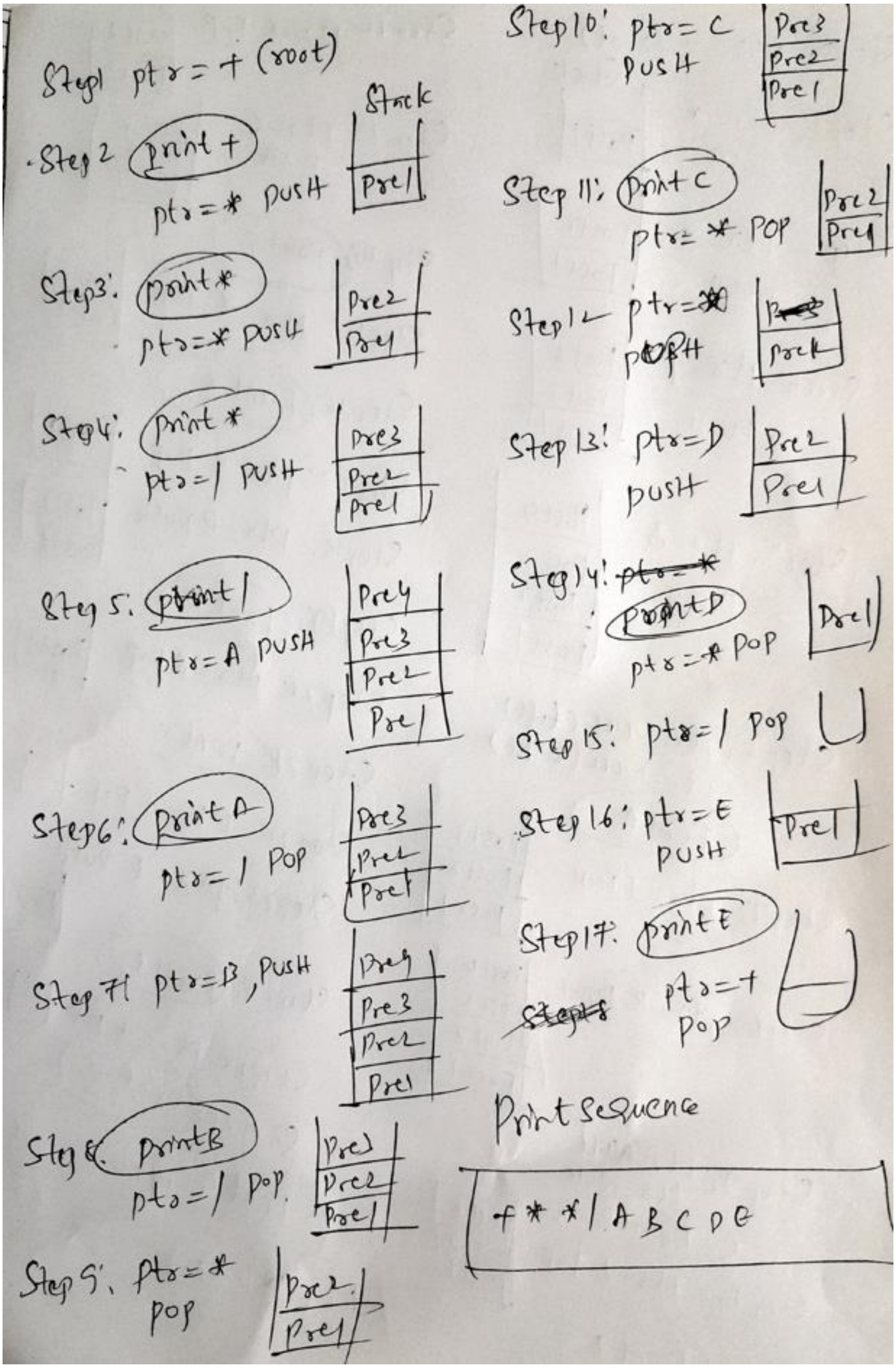
(ii) Preorder:

In the preorder traversal, node is printed in the path it is traversed towards left and right subtrees. Hence technique is NLR (Node Left Right). The C recursive procedure for preorder traversal is:

```
void preorder(treePointer ptr)
{
     if(ptr)
    {
          printf("%d",ptr->data);
          preorder(ptr->leftChild);
          preorder(ptr->rightChild);
    }
}
```

Example preorder traversal for the tree:





Step1 ptr = + (root)

Stack

Step 2 print +
ptr = * PUSH
Pre1

Step3: print *
ptr = * PUSH
Pre2
Pre1

Step4: print *
ptr = / PUSH
Pre3
Pre2
Pre1

Step 5: print /
ptr = A PUSH
Pre4
Pre3
Pre2
Pre1

Step6: print A
ptr = / POP
Pre3
Pre2
Pre1

Step 7: ptr = B, PUSH
Pre4
Pre3
Pre2
Pre1

Step 8: print B
ptr = / Pop
Pre3
Pre2
Pre1

Step 9: ptr = *
pop
Pre2
Pre1

Step10: ptr = C
PUSH
Pre3
Pre2
Pre1

Step 11: print C
ptr = * POP
Pre2
Pre1

Step 12 ptr = [illegible]
POP
~~Pre2~~
Pre1

Step 13: ptr = D
PUSH
Pre2
Pre1

Step 14: ~~ptr = *~~
print D
ptr = * POP
Pre1

Step 15: ptr = / Pop

Step 16: ptr = E
PUSH
Pre1

Step 17: print E
~~Step 18~~
ptr = +
pop

Print sequence

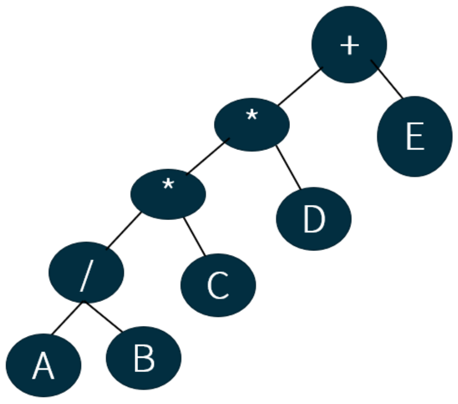
+ * * / A B C D E

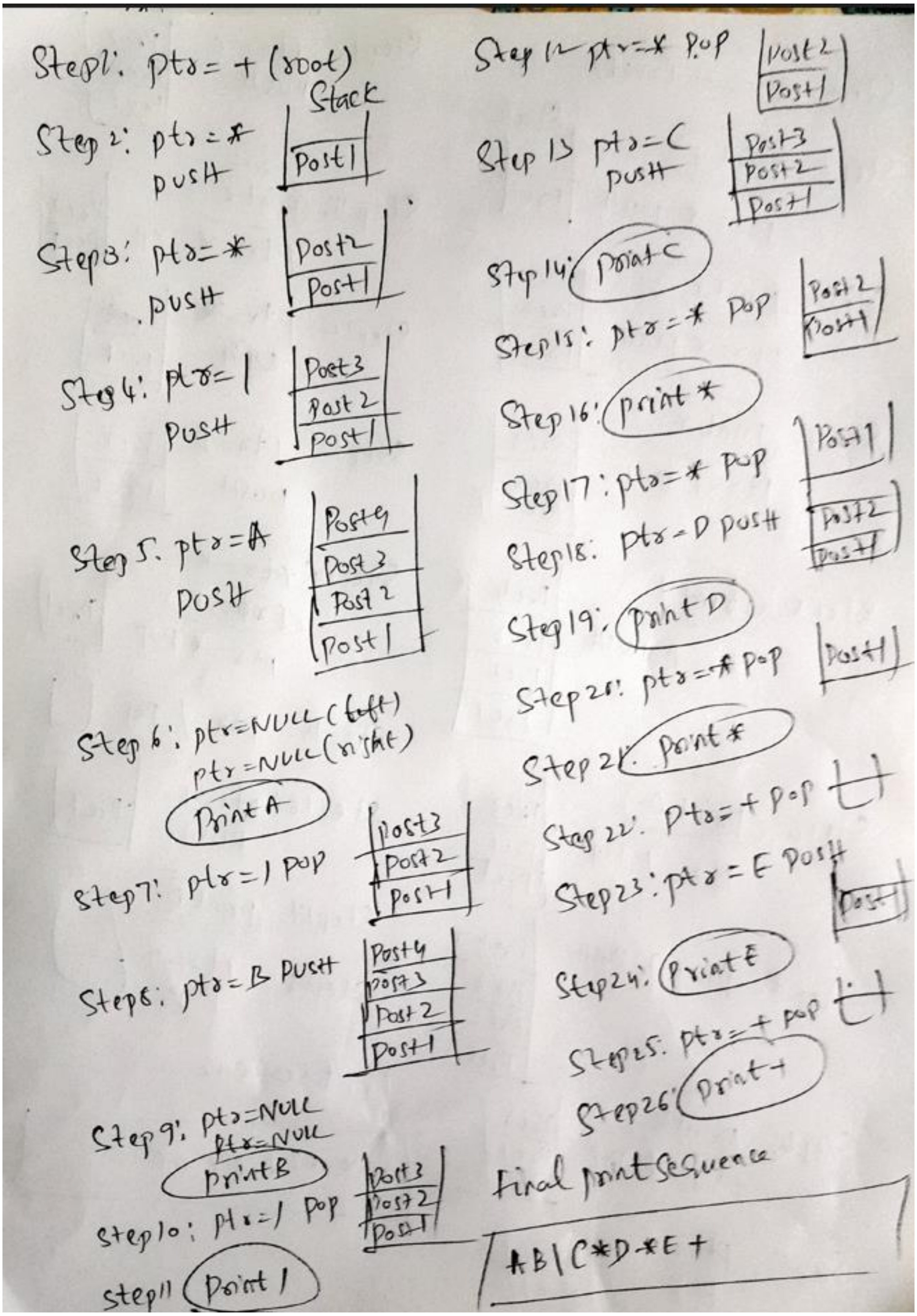
(iii) Post order:

In the postorder traversal, left and right subtrees are completely traversed and then nodes are printed. Hence technique is LRN (Left Right Node). The C recursive procedure for postorder traversal is:

```
void postorder(treePointer ptr)
{
     if(ptr)
    {
          postorder(ptr->leftChild);
          postorder(ptr->rightChild);
          printf("%d",ptr->data);
    }
}
```

Example postorder traversal for the tree:





Step1: ptr = + (root)

Stack

Step 2: ptr = *
push

Post1

Step3: ptr = *
push

Post2
Post1

Step4: ptr = |
push

Post3
Post 2
post1

Step 5. ptr = A
push

Post4
Post 3
Post 2
Post 1

Step 6: ptr=NULL (left)
ptr=NULL (right)
Print A

step7: ptr = | pop

Post3
Post2
post1

Step8: ptr = B push

Post4
Post3
Post 2
Post1

Step 9: ptr=NULL
~~ptr=NULL~~
printB

step10: ptr = | pop

Post3
Post2
Post1

step11 Print |

Step 12 ptr = * Pop

Post2
Post1

Step 13 ptr = C
push

Post3
Post2
Post1

Step 14: Print C

Step15: ptr = * Pop

Post 2
Post1

Step 16: print *

Step17: ptr = * pop

Post1

Step18: ptr = D push

Post2
Post1

Step19: print D

Step 20: ptr = * pop

Post1

Step 21. Print *

Step 22: ptr = + pop

Step23: ptr = E push

Post1

Step24: Print E

Step25. ptr = + pop

Step26 Print +

Final print sequence

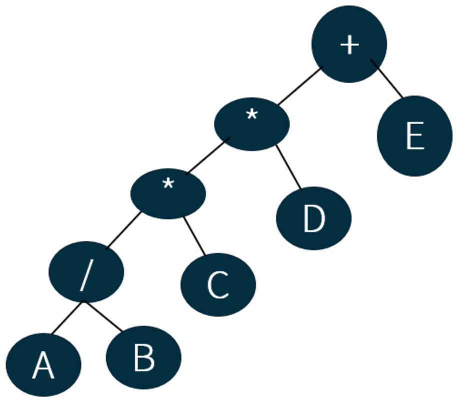
A B | C * D * E +

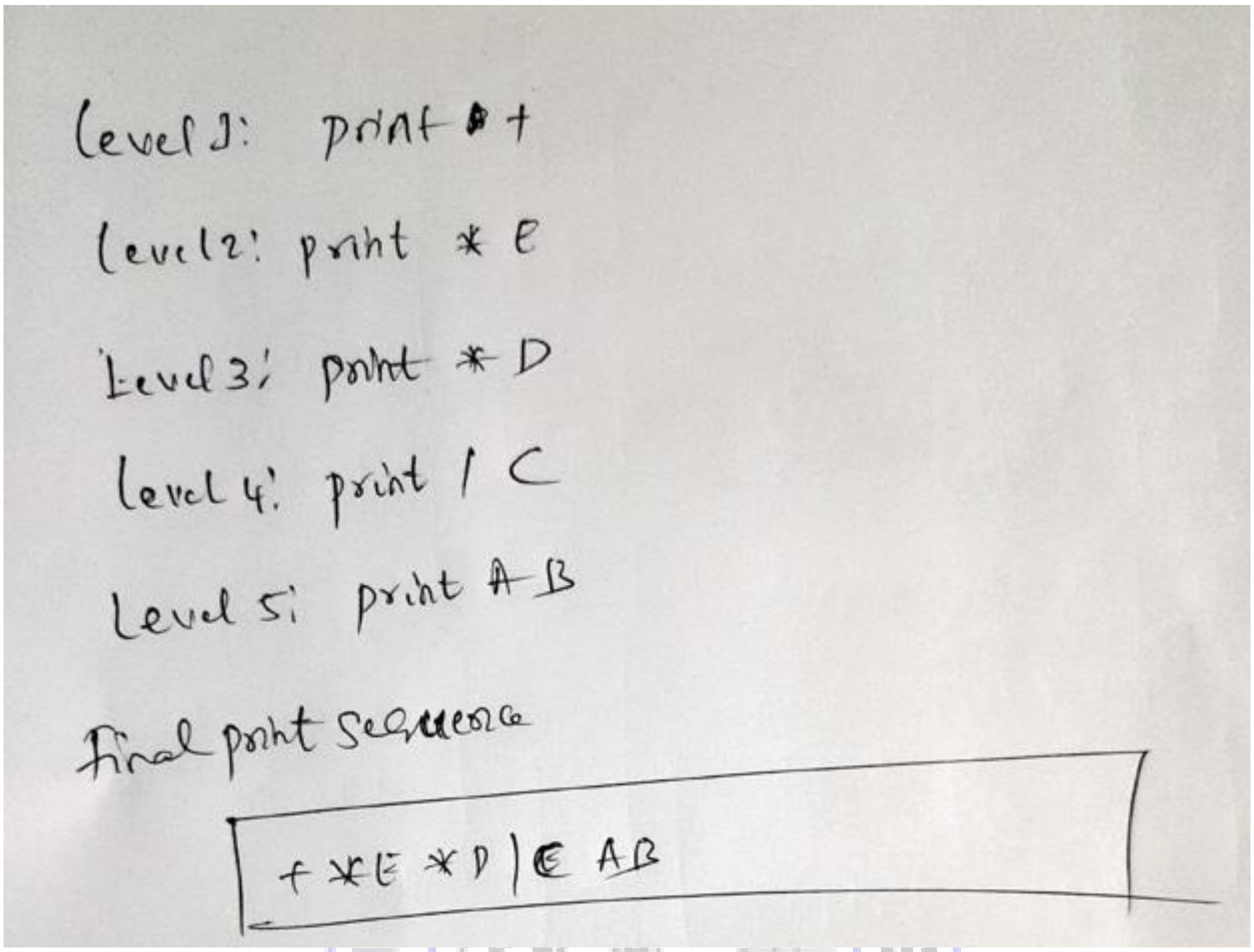
Level-order traversal:

In the level order traversal of a tree, the nodes are visited level by level starting from level-1. In each level, nodes are visited left to right. C procedure for level-order traversal is:

```
void levelorder(treePointer ptr)
{
     treePointer queue[MAX_SIZE];
      if(!ptr) return;
       addq(ptr);
        for(;;)
       {
              ptr=deleteq();
              if(ptr)
               {
                   printf("%d",ptr->data);
                   if(ptr->leftChild)
                           addq(ptr->leftChild);
                   if(ptr->rightChild)
                           addq(ptr->rightChild);
              }
else
    break;
}
}
```

Example level-order traversal for the tree:





level 1: print +

level 2: print * E

Level 3: print * D

level 4: print / C

Level 5: print A B

Final print sequence

+ *E *D / C AB