

LECTURE NOTES

ON

DATA STRUCTURES

ACADEMIC YEAR 2021-22

I B.Tech –II SEMESTER (R20)

M.V.Ramana ,Associate Professor



DEPARTMENT OF HUMANITIES AND BASIC SCIENCES

V S M COLLEGE OF ENGINEERING

RAMCHANDRAPURAM

E.G DISTRICT - 533255

VSM COLLEGE OF ENGINEERING
RAMACHANDRAPRUM-533255
DEPARTMENT OF HUMANITIES AND BASIC SCIENCES

Course Title	Year-Sem	Branch	Contact Periods/Week	Sections
Data Structures	1-2	Computer Science and Engineering	5	-

COURSE OUTCOMES: After completing this course a student will be able to:

- Summarize the properties, interfaces, and behaviors of basic abstract data types
- Discuss the computational efficiency of the principal algorithms for sorting & searching
- Use arrays, records, linked structures, stacks, queues, trees, and Graphs in writing programs
- Demonstrate different methods for traversing trees

Unit/Item No.	Outcomes	Topic	Number of periods	Total periods	Book Reference	Delivery Method	
1	CO 2: Discuss the computational efficiency of the principal algorithms for sorting & searching	Data Structures, Sorting and Searching		16	T1, T2, R1	Chalk & Talk	
		1.1	Data Structures - Definition, Classification of Data Structures				2
		1.2	Operations on Data Structures, Abstract Data Type (ADT)				2
		1.3	Preliminaries of algorithms. Time and Space complexity				2
		1.4	Linear search, Binary search				1
		1.5	Fibonacci search				1
		1.6	Insertion sort, Selection sort				2
		1.7	Bubble sort, quick sort				2
		1.8	Problems on above topics				2
		1.9	Radix Sort and Merge Sort				2
2	CO 3: Use arrays, records, linked structures, stacks, queues, trees, and Graphs in writing programs	Linked List		14	T1, T2, R1	Chalk & Talk	
		2.1	Introduction, Single linked list, Representation of Linked list				1
		2.2	Operations on Single Linked list- Insertion, Deletion, Search and Traversal				2
		2.3	Reversing Single Linked list, Applications on SLL				1
		2.4	Polynomial Expression Representation				2
		2.5	Addition of Polynomials				1
		2.6	Multiplication of Polynomials				1
		2.7	Sparse Matrix Representation				1
2.8	Advantages and Disadvantages of	1					

			Single Linked list				
		2.9	Double Linked list-Insertion, Deletion, Circular Linked list-Insertion, Deletion	4			
		Stacks and Queues					
		3.1	Introduction to Queues, Representation of Queues-using Arrays and using Linked list	2			
		3.2	Implementation of Queues-using Arrays	1			
		3.3	Implementation of Queues-using Linked List	1			
		3.4	Application of Queues Circular Queues, Deques, Priority Queues, Multiple Queues	2			
		3.5	Introduction to Stacks, Array Representation of Stacks	1			
		3.6	Operations on Stacks, Linked list Representation of Stacks	2			
		3.7	Operations on Linked Stack, Applications	2			
		3.8	Infix to Postfix Conversion, Evaluating Postfix Expressions	3			
3	CO 3: Use arrays, records, linked structures, stacks, queues, trees, and Graphs in writing programs				14	T1,T2, R2	Chalk & Talk
		Trees					
		4.1	Basic Terminology in Trees	1			
		4.2	Binary Trees Properties, Representation of Binary Trees using Arrays and Linked lists	2			
		4.3	Binary Search Trees- Basic Concepts, BST Operations: Insertion, Deletion, Tree Traversals	3			
		4.4	Heap Tree and Heap Sort	2			
		4.5	Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations	2			
4	CO4: Demonstrate different methods for traversing trees				10	T1, R1	Chalk & Talk,ppt
		Graphs					
		5.1	Basic Concepts, Representations of Graphs	1			
		5.2	Graph Traversals (BFT & DFT)	1			
		5.3	Applications- Minimum Spanning Tree Using Prim's & Kruskal's Algorithm	1			
		5.4	Dijkstra's shortest path	1			
		5.5	Transitive closure, Warshall's Algorithm	2			
5	CO 3: Use arrays, records, linked structures, stacks, queues, trees, and Graphs in writing programs				6	T3, R7	Chalk & Talk
				TOTAL	60		

LIST OF TEXT BOOKS AND AUTHORS

Text Books:

- 1) Data Structures Using C. 2nd Edition. Reema Thareja, Oxford.
- 2) Data Structures and algorithm analysis in C, 2nd ed, Mark Allen Weiss.

Reference Books:

- 1) Fundamentals of Data Structures in C, 2nd Edition, Horowitz, Sahni, Universities Press.
- 2) Data Structures: A PseudoCode Approach, 2/e, Richard F. Gilberg, Behrouz A. Forouzon, Cengage.
- 3) Data Structures with C, Seymour Lipschutz TMH



Faculty Member



Head of the Department



PRINCIPAL



JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

I Year – II Semester	L	T	P	C
	3	0	0	3
DATA STRUCTURES				

Course Objectives:

The objective of the course is to

- Introduce the fundamental concept of data structures and abstract data types
- Emphasize the importance of data structures in developing and implementing efficient algorithms
- Describe how arrays, records, linked structures, stacks, queues, trees, and graphs are represented in memory and used by algorithms

Course Outcomes:

After completing this course a student will be able to:

- Summarize the properties, interfaces, and behaviors of basic abstract data types
- Discuss the computational efficiency of the principal algorithms for sorting & searching
- Use arrays, records, linked structures, stacks, queues, trees, and Graphs in writing programs
- Demonstrate different methods for traversing trees

UNIT I

Data Structures - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity.

Searching - Linear search, Binary search, Fibonacci search.

Sorting- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

UNIT II

Linked List: Introduction, Single linked list, Representation of Linked list in memory, Operations on Single Linked list-Insertion, Deletion, Search and Traversal ,Reversing Single Linked list, Applications on Single Linked list- Polynomial Expression Representation ,Addition and Multiplication, Sparse Matrix Representation using Linked List, Advantages and Disadvantages of Single Linked list, Double Linked list-Insertion, Deletion, Circular Linked list-Insertion, Deletion.

UNIT III

Queues: Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.

Stacks: Introduction to Stacks, Array Representation of Stacks, Operations on Stacks, Linked list Representation of Stacks, Operations on Linked Stack, Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

UNIT IV

Trees: Basic Terminology in Trees, Binary Trees-Properties, Representation of Binary Trees using Arrays and Linked lists. Binary Search Trees- Basic Concepts, BST Operations: Insertion, Deletion, Tree Traversals, Applications-Expression Trees, Heap Sort, Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations.



JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIT V

Graphs: Basic Concepts, Representations of Graphs-Adjacency Matrix and using Linked list, Graph Traversals (BFT & DFT), Applications- Minimum Spanning Tree Using Prim's & Kruskal's Algorithm, Dijkstra's shortest path, Transitive closure, Warshall's Algorithm.

Text Books:

- 1) Data Structures Using C. 2nd Edition. Reema Thareja, Oxford.
- 2) Data Structures and algorithm analysis in C, 2nded, Mark Allen Weiss.

Reference Books:

- 1) Fundamentals of Data Structures in C, 2nd Edition, Horowitz, Sahni, Universities Press.
- 2) Data Structures: A PseudoCode Approach, 2/e, Richard F. Gilberg, Behrouz A. Forouzon, Cengage.
- 3) Data Structures with C, Seymour Lipschutz TMH

e-Resources:

- 1) <http://algs4.cs.princeton.edu/home/>
- 2) https://faculty.washington.edu/jstraub/dsa/Master_2_7a.pdf

Unit – I

Syllabus:

- Data Structures - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity.
- Searching - Linear search, Binary search, Fibonacci search.
- Sorting- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

INTRODUCTION:

- A *data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables
- Today computer programmers do not write programs just to solve a problem but to write an efficient program.
- When selecting a data structure to solve a problem, the following steps must be performed.
 1. Analysis of the problem to determine the basic operations that must be supported.
 2. Quantify the resource constraints for each operation.
 3. Select the data structure that best meets these requirements.
- The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of π , etc.).
- A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.
- A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file.

CLASSIFICATION OF DATA STRUCTURES:

- Data structures are generally categorized into two classes: *primitive* and *non-primitive* data structures.

Primitive and Non-primitive Data Structures:

- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type, basic data type’, and ‘primitive data type’ are often used interchangeably.

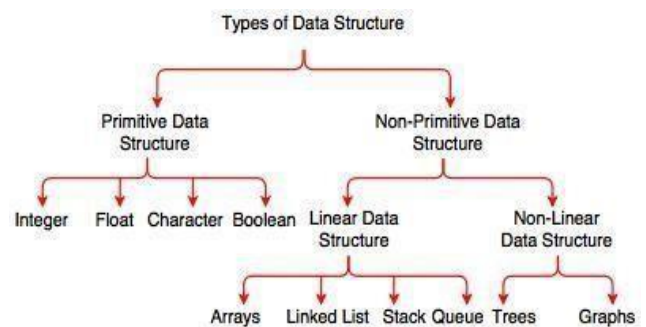


Fig. Types of Data Structure

- Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.
- Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

Linear and Non-linear Structures:

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
 - Examples include arrays, linked lists, stacks, and queues.
 - Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.
- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.
 - The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Arrays:

- An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).
- In C, arrays are declared using the following syntax: datatype name[size];
Ex: int marks[10];

1 st element	2 nd element	3 rd element	4 th element	5 th element	6 th element	7 th element	8 th element	9 th element	10 th element
----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	-----------------------------

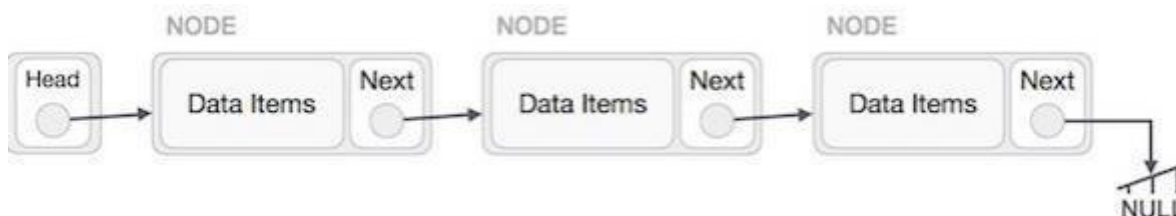
marks[0] marks[1] marks[2] marks[3] marks[4] marks[5] marks[6] marks[7] marks[8] marks[9]

limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

Linked Lists:

- linked list is a dynamic data structure in which elements (called *nodes*) form a sequential list.
- In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list.
- Every node contains the following
 - The value of the node or any other data that corresponds to that node
 - A pointer or link to the next node in the list



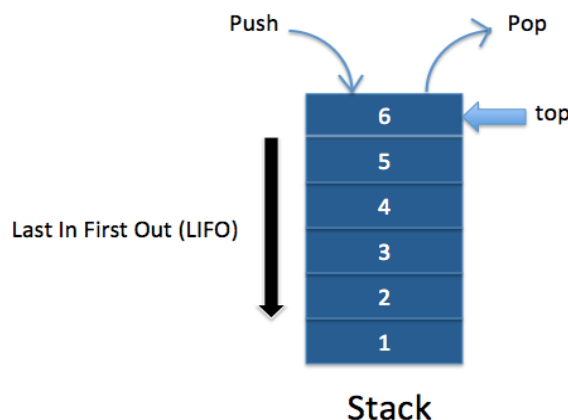
- The first node in the list is pointed by Head/Start/First. The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

Advantage: Easier to insert or delete data elements

Disadvantage: Slow search operation and requires more memory space

Stacks:

- A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.
- Stacks can be implemented using arrays or linked lists.
- Every stack has a variable top associated with it. Top is used to store the address of the topmost element of the stack.
- It is this position from where the element will be added or deleted. There is another variable MAX, which is used to store the maximum number of elements that the stack can store.
- If $top = NULL$, then it indicates that the stack is empty and if $top = MAX - 1$, then the stack is full.
- A stack supports three basic operations: push, pop, and peep. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the peep operation returns the value of the topmost element of the stack (without deleting it).



Queues:

- A Queue is a linear data structure in which insertion can be done at rear end and deletion of elements can be done at front end.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- Like stacks, queues can be implemented by using either arrays or linked lists.



Front						Rear			
12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Insert element into the Queue:

Front						Rear			
12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

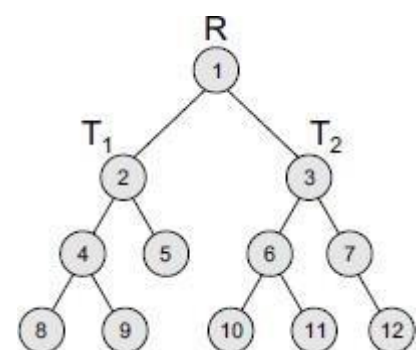
Delete element from Queue:

Front					Rear				
	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

- A queue is full when $\text{rear} = \text{MAX} - 1$, An underflow condition occurs when we try to delete an element from a queue that is already empty. If $\text{front} = \text{NULL}$ and $\text{rear} = \text{NULL}$, then there is no element in the queue.

Trees:

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.
- The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.
- Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree.
- The root element is the topmost node which is pointed by a 'root' pointer. If $\text{root} = \text{NULL}$ then the tree is empty.



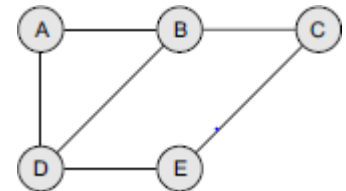
- Here R is the root node and T1 and T2 are the left and right subtrees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty, then it is called the right successor of R.

Advantage: Provides quick search, insert, and delete operations

Disadvantage: Complicated deletion algorithm

Graphs:

- A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- A node in the graph may represent a city and the edges connecting the nodes can represent roads.
- A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.
- Graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph.



Advantage: Best models real-world situations

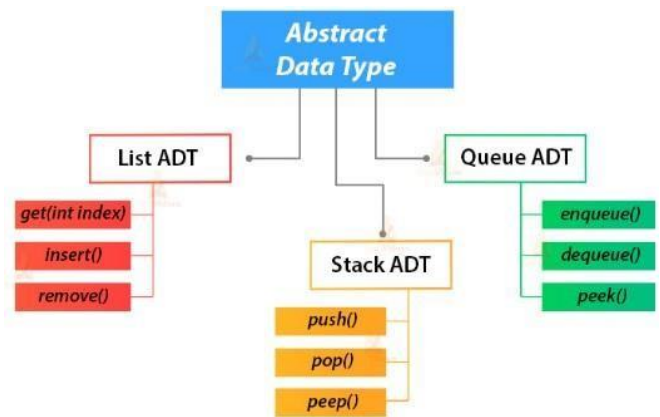
Disadvantage: Some algorithms are slow and very complex

OPERATIONS ON DATA STRUCTURES:

- This section discusses the different operations that can be performed on the various data structures previously mentioned.
- **Traversing** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.
- **Searching** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.
- **Inserting** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
- **Deleting** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.
- **Sorting** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.
- **Merging** Lists of two sorted data items can be combined to form a single list of sorted data items.

ABSTRACT DATA TYPE:

- An *abstract data type* (ADT) is a data structure, focusing on what it does and ignoring how it does its job. (or) Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- Ex: stacks ADT and queues ADT. the user is concerned only with the type of data and the operations that can be performed on it.
- We can implement both these ADTs using an array or a linked list.



Advantage of using ADTs

- Modification of a program is simple, For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.
- In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation.

PRELIMINARIES OF ALGORITHM:

- Algorithm is step by step logical procedure for solving a problem.
- In Algorithm each step is called *Instruction*.
- An Algorithm is any well-defined computational procedure that take some values as inputs and produce some values as output.
- An Algorithm is a sequence of computational steps that transform input into output.
- An Algorithm has 5 basic properties:
 1. *Input*: An Algorithm has take '0' or more number of inputs that can be supplied as externally.
 2. *Output*: An Algorithm must produce at least one output.
 3. *Definiteness*: Each instruction in the algorithm must be clear.
 4. *Finiteness*: An algorithm must terminate after a finite number of steps.
 5. *Effectiveness*: Each operation should be effective. i.e the operations must be terminate after finite amount of time.

Structure of an Algorithm:

1. Algorithm is a procedure consisting of heading and body. In body part we are writing statements and in the head part we are writing the following.

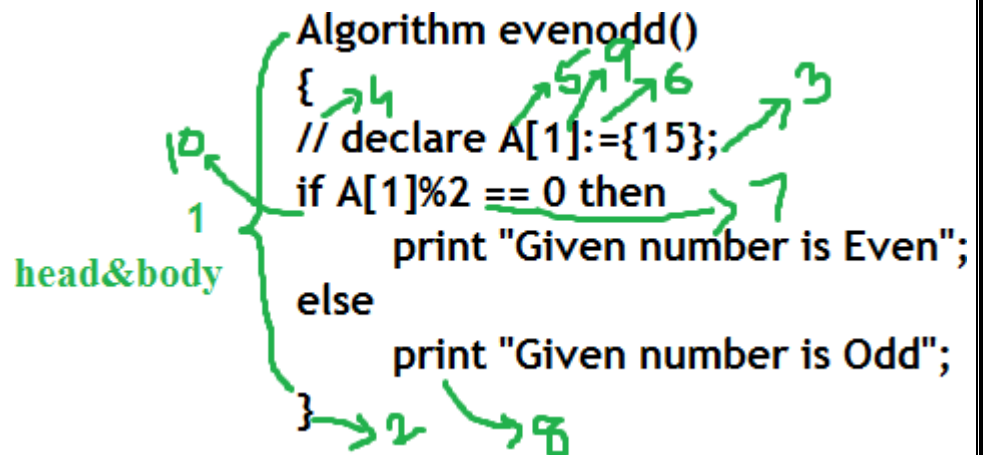
Syntax: Algorithm name_of_Algo (param1,param2, ...);

2. The beginning and ending of block should be indicated by '{' and '}' or 'start' and 'end' respectively.

3. Every statement in the algorithm should be end with semicolon (;).

4. Single line comments are written using '//' as beginning of comments.

5. The identifier should begin with character and it may be combination of alpha numeric.



6. Assignment operator (:=) we can use as follows

Variable := expression (or) value;

7. There are other type of operators such as Boolean operators (TRUE/FALSE), logical operators (AND,OR,NOT) and relational operators (<,>,<=,>=,.....)

8. The input and output we can write it as read and print respectively.

9. The Array index are stored with in [] brackets. The index of array starts from '0' to 'N-1'.

Syntax: datatype Aray_name[size];

10. The conditional statements such as if-then (or) if-then-else are written as follows.

if(condition) then statements;

if(condition) then

statements;

else

statements;

TIME AND SPACE COMPLEXITY:

- The efficiency of an algorithm can be computed by measuring the performance of an algorithm.

We can measure the performance of an algorithm in Two(2) ways.

1. Time Complexity
2. Space Complexity

1. Time Complexity:

- The time complexity of an algorithm is the amount of computing time required by an algorithm to run its completion.
- There are 2 types of computing time 1. Compile time 2. Run time

- The time complexity generally computed at run time (or) execution time.
- The time complexity can be calculated in terms of frequency count.
- Frequency count is a count denoting the number of times the statement should be executed.
- The time complexity can be calculated as

Comments – 0

Assignment / return statement – 1

Conditional (or) Selection Constructs – 1

Example 1: Sum of the elements in an Array

Statements	Step count/ Execution	Frequency	Total Steps
Algorithm Addition (A,n)	0	-	0
{	0	-	0
//A is an array of size 'n'	0	-	0
Sum :=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
Sum:=Sum+A[i];	1	n	n
return Sum;	1	1	1
}	0	-	0
Total		2n+3	

Example 2: Subtraction of two matrices

Statements	Step count/ Execution	Frequency	Total Steps
Algorithm Subtract (A,B,C,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
C[i,j] := A[i,j] – B[i,j];	1	mn	mn
}	0	-	0
Total		2mn+2m+1	

2. Space Complexity:

- Space Complexity can be defined as amount of memory (or) space required by an Algorithm to run.
- To compute the space complexity we use 2 factors i. Constant ii. Instance characteristics.
- The space requirement $S(p)$ can be given as $S(p) = C + S_p$

Where C- Constant, it denotes the space taken for input and output.

S_p – Amount of space taken by an instruction, variable and identifiers.

Example 1: Sum of three numbers

```
Algorithm Add(a,b,c)
{
//a,b,c are float type variables
return a+b+c;
}
```

- The space required for this algorithm is: Assume a,b,c are occupies 1 word size each, total size comes to be **3**.

Example 2: Sum of Array values

```
Algorithm Addition (A,n)
{
//A is an array of size 'n'
Sum :=0;
for i:=1 to n do
    Sum:=Sum+A[i];
return Sum;
}
```

- The space required for this algorithm is:
One word space for each variable then i,sum,n \rightarrow 3
For Array A[] we require the size \rightarrow n
Total space complexity for this algorithm is **S(p) \geq (n+3)**

What to Analyze in an algorithm:

An Algorithm can require different times to solve different problems of same size

1. Worst case: Maximum amount of time that an algorithm require to solve a problem of size 'n'.
Normally we can take upper bound as complexity. We try to find worst case behavior.
2. Best case: Minimum amount of time that an algorithm require to solve a problem of size 'n'.
Normally it is not much useful.
3. Average case: the average amount of time that an algorithm require to solve a problem of size 'n'.
Some times it is difficult to find. Because we have to check all possible data organizations

SEARCHING:

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- Searching techniques are *linear search*, *binary search* and *Fibonacci Search*

LINEAR SEARCH:

- Linear search is a technique which traverse the array sequentially to locate given item or search element.
- In Linear search, we access each element of an array one by one sequentially and see weather it is desired element or not. We traverse the entire list and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.
- A search is successful then it will return the location of desired element
- If A search will unsuccessful if all the elements are accessed and desired element not found.
- Linear search is mostly used to search an unordered list in which the items are not sorted.

Linear search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Example:

Consider the following list of elements and the element to be searched...

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

BINARY SEARCH:

- Binary search is the search technique which works efficiently on the **sorted lists**. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.
- Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example:

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are matching. So the result is "Element found at index 1"

Example 2:

search element 80

Step 1:

search element (80) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. So the result is "Element found at index 7"

FIBONACCI SEARCH:

- **Fibonacci search** is an efficient search algorithm based on **divide and conquer** principle that can find an element in the given **sorted array** with the help of Fibonacci series in **O(log N)** time complexity. This is based on Fibonacci series which is an infinite sequence of numbers denoting a pattern which is captured by the following equation:

$$\begin{array}{ll} \mathbf{F(n)=n} & \mathbf{if\ n\leq 1} \\ \mathbf{F(n)=F(n-1)+F(n-2)} & \mathbf{if\ n>1} \end{array}$$

- where F(i) is the ith number of the Fibonacci series where F(0) and F(1) are defined as 0 and 1 respectively.
- The first few Fibonacci numbers are: **0,1,1,2,3,5,8,13....**

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = F(1) + F(0) = 1 + 0 = 1$$

$$F(3) = F(2) + F(1) = 1 + 1 = 2$$

$$F(4) = F(3) + F(2) = 1 + 2 = 3 \text{ and so continues the series}$$

- Other searches like binary search also work for the similar principle on splitting the search space to a smaller space but what makes Fibonacci search different is that it divides the array in **unequal parts** and operations involved in this search are **addition and subtraction** these arithmetic operations takes place simple and hence **reducing the work load of the computing machine**.

Algorithm:

- Let the length of given array be **n [0. n-1]** and the element to be searched be **x**.
- Then we use the following steps to find the element with minimum steps:

1. Find the smallest Fibonacci number greater than or equal to n. Let this number be **f(M)**

Let the two Fibonacci numbers preceding it be **f(M-1)** and **f(M-2)**.

$$F(M) = F(\text{Size of array})$$

$$F(M-1) = F(M) - 1$$

$$F(M-2) = F(M-1) - 1$$

$$i(\text{index}) = \min(\text{offset} + F(M-2), n-1) // \text{Offset} = -1$$

2. While the array has elements to be checked:

-> Compare x with the last element of the range covered by f(M-2)

-> If **x** matches, return index value

-> Else if **x is less** than the element, move the three Fibonacci variables two Fibonacci down, Indicating removal of approximately two-third of the unsearched array from rear end. Not Reset offset to index

-> Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Indicating removal of approximately one-third of the unsearched array from front end.

3. Since there might be a single element remaining for comparison, check if F(M-1) is '1'. If Yes, compare x with that remaining element. If match, return index value.

Example: The Elements in array & Search key is

Search_Key 85

elements	10	22	35	40	45	50	80	82	85	90	95
Index	0	1	2	3	4	5	6	7	8	9	10

Initially the Fibonacci series is ...

0	1	1	2	3	5	8	13	21	34
1	2	3	4	5	6	7	8	9	10
					F(m-2)	F(m-1)	F(m)		

To calculate index position $i = \min(\text{offset} + F(m-2), n-1)$, Initially offset value is -1.

F(m)	F(m-1)	F(m-2)	Offset	i(index)	a[i]	Consequence
13	8	5	-1	$(-1+5, 10) = 4$	45	1 steps down, Reset offset
8	5	3	4	$(4+3, 10) = 7$	82	1 steps down, Reset offset
5	3	2	7	$(7+2, 10) = 9$	90	2 steps down
2	1	1	7	$(7+1, 10) = 8$	85	Return i

Finally our desired element is **found at the location of 8.**

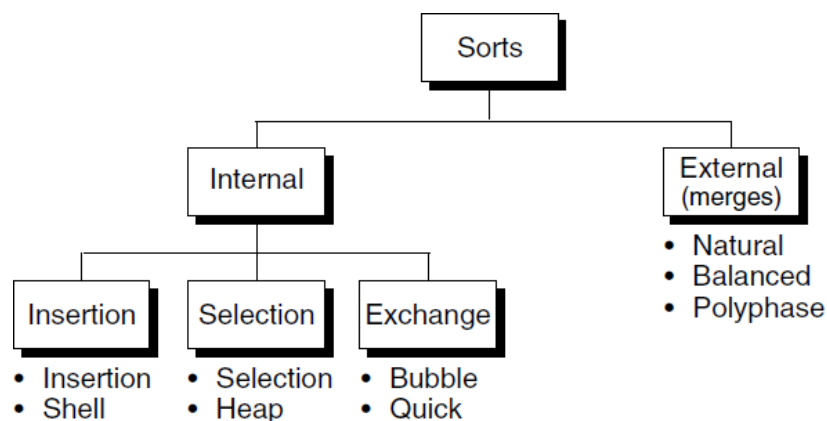
SORTINGS:

- **Definition:** Sorting is a technique to rearrange the list of records(elements) either in ascending or descending order, Sorting is performed according to some key value of each record.

Categories of Sorting:

The sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

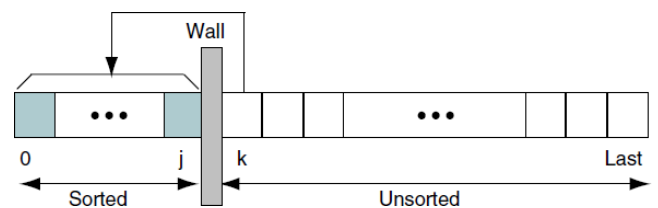


- **Internal Sorting:** When all the data that is to be sorted can be accommodated at a time in the main memory (Usually RAM). Internal sortings has five different classifications: insertion, selection, exchanging, merging, and distribution sort
- **External Sorting:** When all the data that is to be sorted can't be accommodated in the memory (Usually RAM) at the same time and some have to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc.

Ex: Natural, Balanced, and Polyphase.

INSERTION SORT:

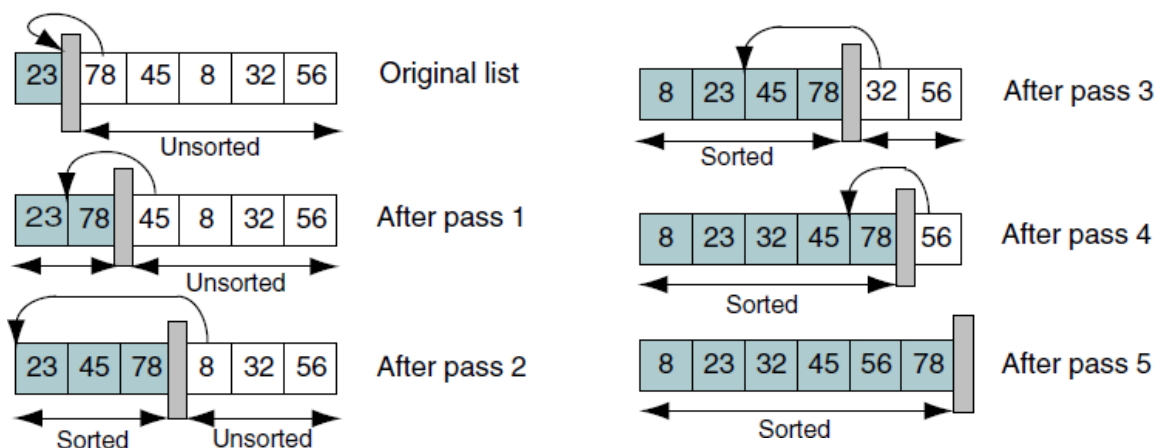
- In Insertion sort the list can be divided into two parts, one is sorted list and other is unsorted list. In each pass the first element of unsorted list is transfers to sorted list by inserting it in appropriate position or proper place.
- The similarity can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



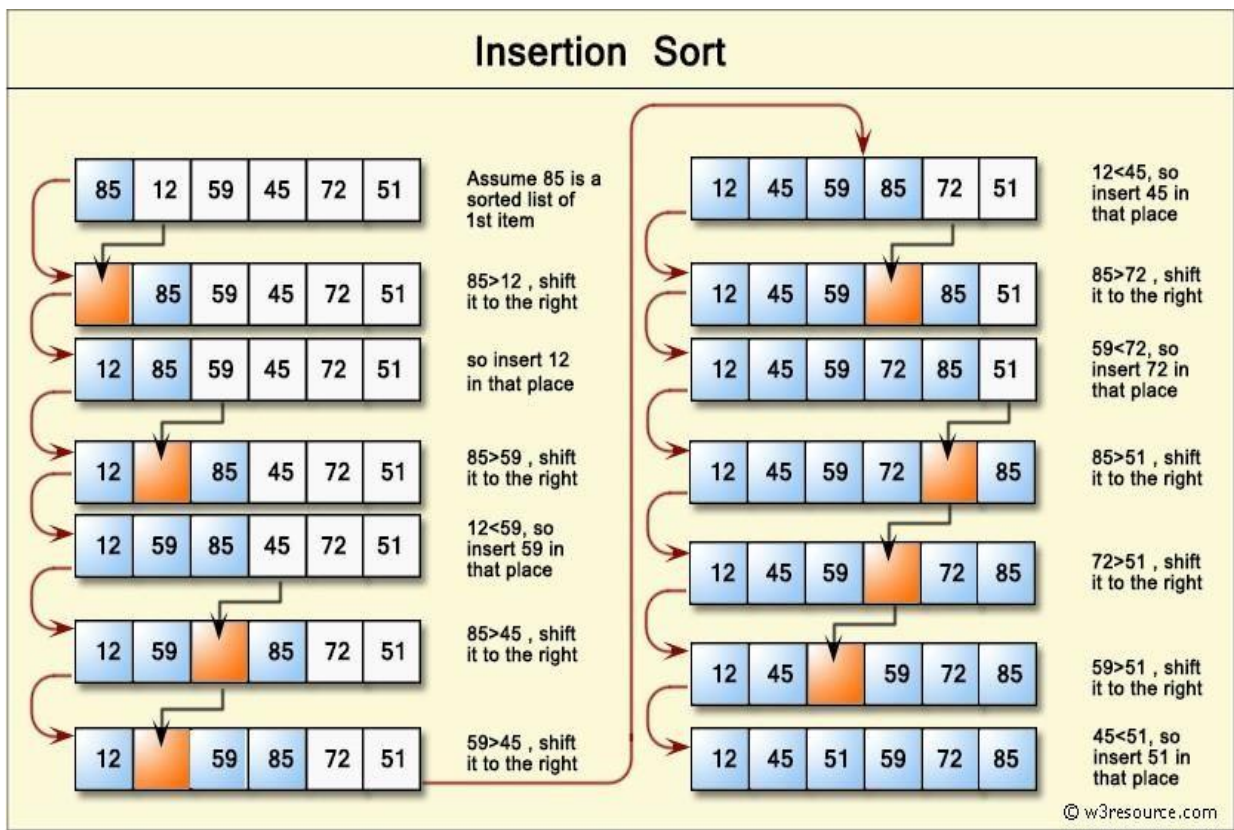
Following are the steps involved in insertion sort:

1. We start by taking the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards
2. We compare the key element with the element(s) before it, in this case, element at index 0:
 - If the key element is less than the first element, we insert the key element before the first element.
 - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the proper position.
4. And we go on repeating this, until the array is sorted.

Example 1:

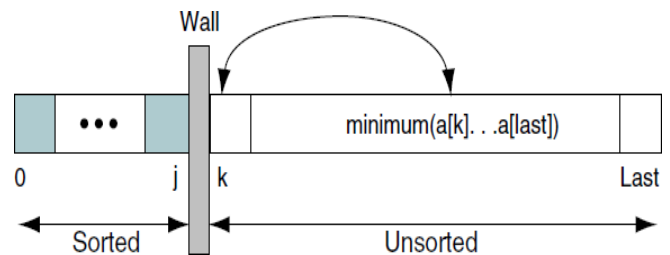


Example 2:



SELECTION SORT:

- Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data.
- In first step, the smallest element is search in the list, once the smallest element is found, it is exchanged with the element in the first position.
- Now the list is divided into two parts. One is sorted list other is unsorted list. Find out the smallest element in the unsorted list and it is exchange with the starting position of unsorted list, after that it will added in to sorted list.
- This process is repeated until all the elements are sorted.



Ex: asked to sort a list on paper.

Algorithm:

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, Loc)

Step 3: SWAP A[K] with ARR[Loc]

Step 4: EXIT

Algorithm for finding minimum element in the list.

SMALLEST (ARR, K, N, Loc)

Step 1: [INITIALIZE] SET Min = ARR[K]

Step 2: [INITIALIZE] SET Loc = K

Step 3: Repeat for J = K+1 to N

IF Min > ARR[J]

SET Min = ARR[J]

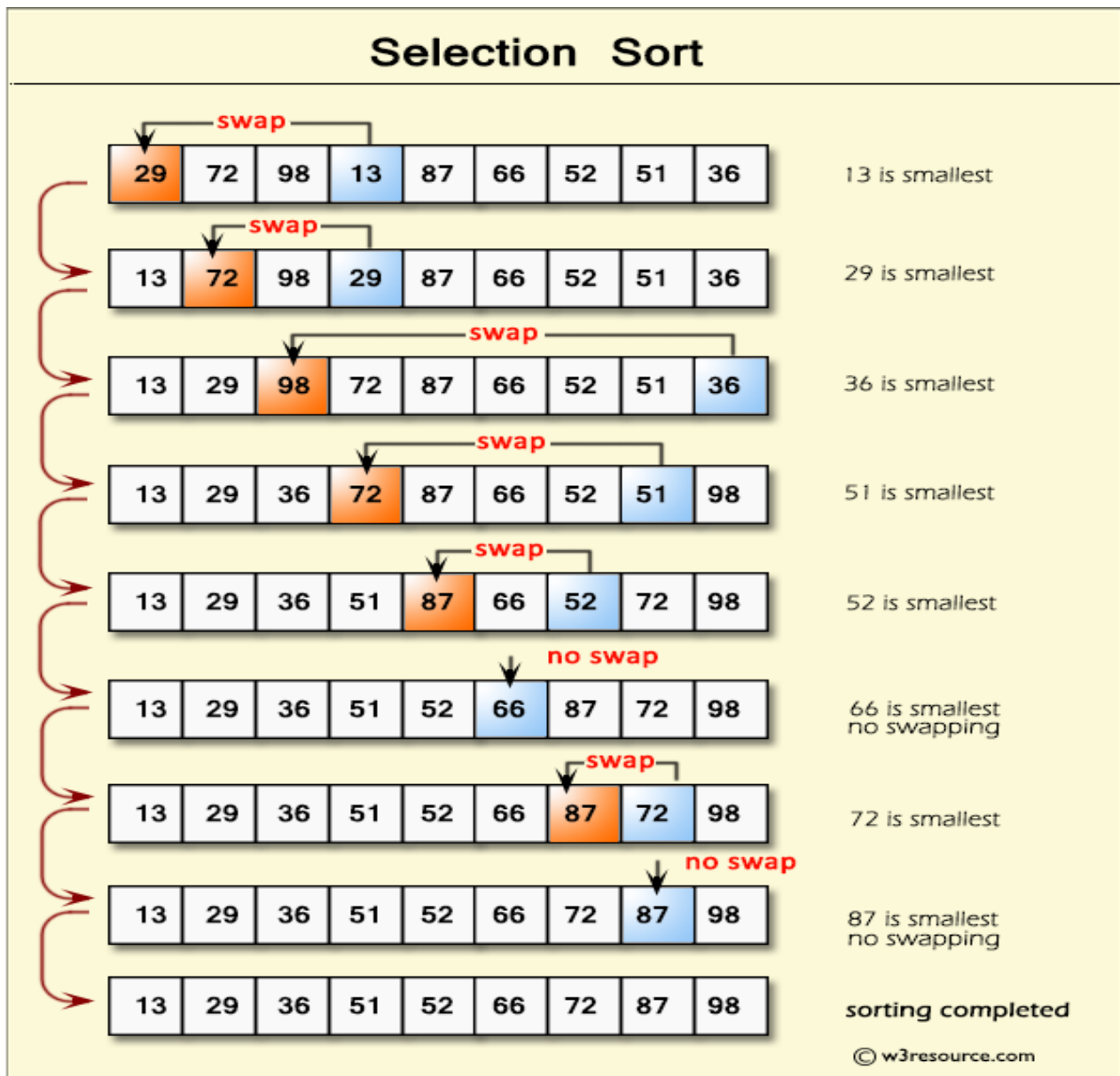
SET Loc = J

[END OF IF]

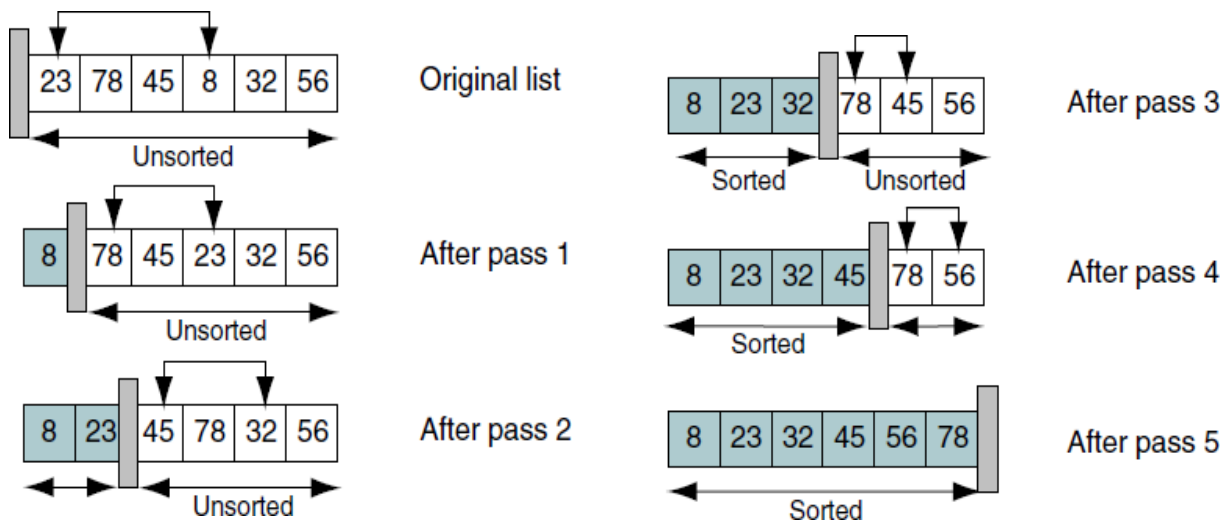
[END OF LOOP]

Step 4: RETURN Loc

Example 1:



Example 2: Consider the elements 23,78,45,88,32,56



Time Complexity:

Number of elements in an array is ‘N’

Number of passes required to sort is ‘N-1’

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 ...

Time required for complete sorting is:

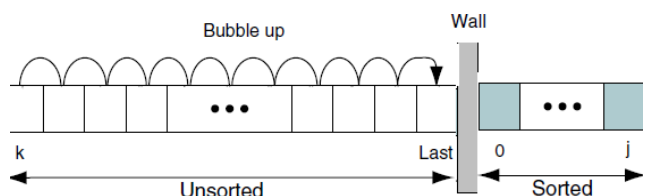
$$T(n) \leq (N-1) * (N-1)$$

$$T(n) \leq (N-1)^2$$

Finally, The time complexity is $O(n^2)$.

BUBBLE SORT:

- Bubble Sort is also called as Exchange Sort
- In Bubble Sort, each element is compared with its adjacent element
 - a) If the first element is larger than the second element then the position of the elements are interchanged.
 - b) Otherwise, the position of the elements are not changed.
 - c) The same procedure is repeated until no more elements are left for comparison.
- After the 1st pass the largest element is placed at (N-1)th location. Given a list of n elements, the bubble sort requires up to $n - 1$ passes to sort the data.



Example 1:

- We take an unsorted array for our example.



- Bubble sort starts with very first two elements, comparing them to check which one is greater.



- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27. We find that 27 is smaller than 33 and these two values must be swapped.



- Next we compare 33 and 35. We find that both are in already sorted positions.



- Then we move to the next two values, 35 and 10. We know then that 10 is smaller 35.



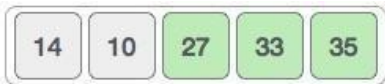
- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



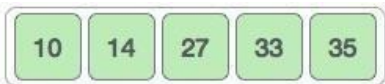
- To be defined, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this



- Notice that after each iteration, at least one value moves at the end.



- And when there's no swap required, bubble sorts learns that an array is completely sorted.



Example 2:



unsorted



5 > 1, swap



5 < 12, ok



12 > -5, swap



12 < 16, ok



1 < 5, ok



5 > -5, swap



5 < 12, ok



1 > -5, swap



1 < 5, ok



-5 < 1, ok



sorted

Algorithm:

BUBBLE SORT(ARR, N)

Step 1: Read the array elements
Step 2: $i:=0$;
Step 3: Repeat step 4 and step 5 until $i<n$
Step 4: $j:=0$;
Step 5: Repeat step 6 until $j<(n-1)-i$
Step 6: if $A[j] > A[j+1]$
 Swap($A[j],A[j+1]$)
 End if
 End loop 5
 End loop 3
Step 7: EXIT

Time Complexity:

Number of elements in an array is 'N'

Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 ...

Time required for complete sorting is:

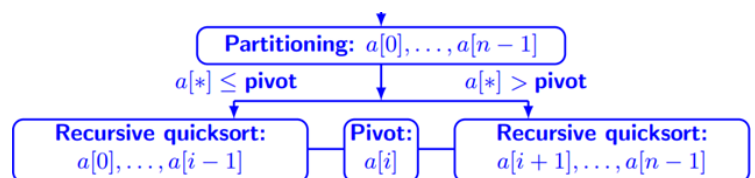
$$T(n) \leq (N-1)*(N-1)$$

$$T(n) \leq (N-1)^2$$

Finally, The time complexity

is $O(n^2)$.

QUICK SORT:



- Quick sort follows **Divide and Conquer** algorithm. It is dividing array in to smaller parts based on partitioning and performing the sort operations on those divided smaller parts. Hence, it works well for large datasets.

So, here are the steps **how Quick sort** works in simple words.

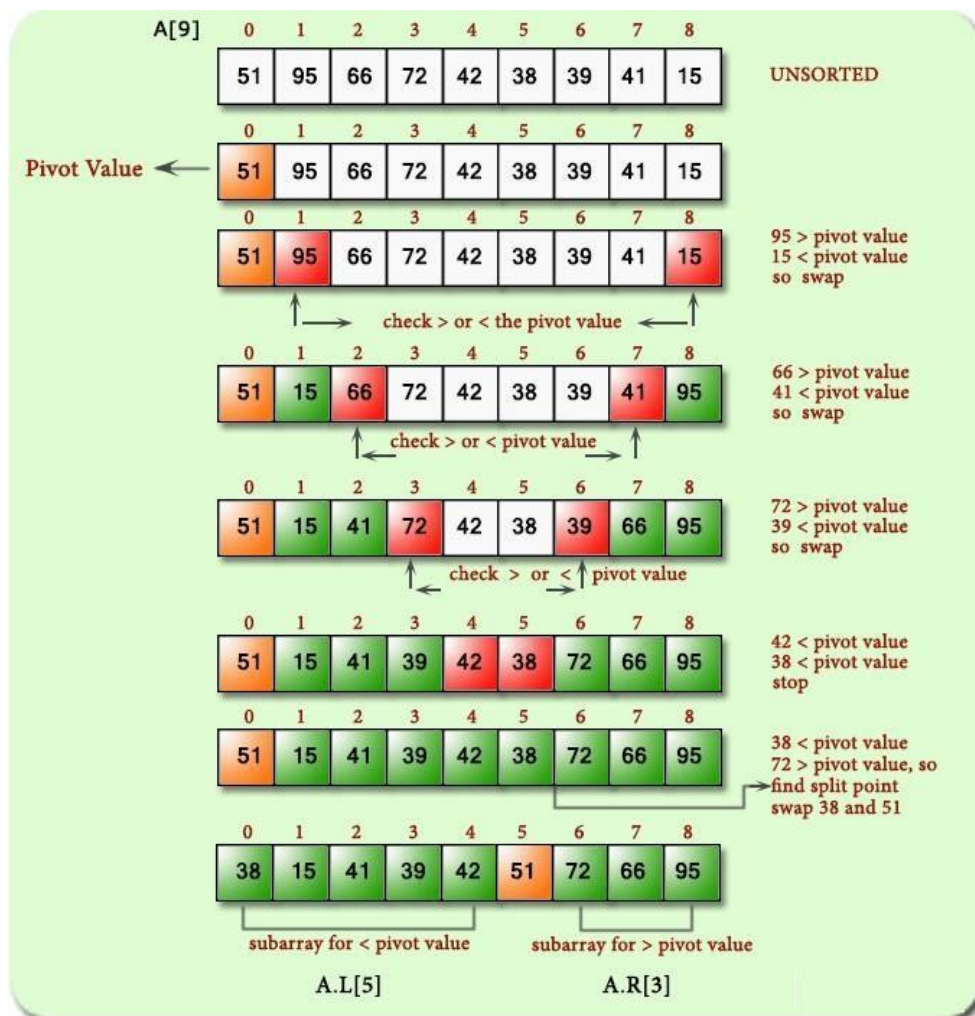
1. First select an element which is to be called as **pivot** element.
2. Next, compare all array elements with the selected pivot element and arrange them in such a way that, elements less than the pivot element are to its left and greater than pivot is to it's right.
3. Finally, perform the same operations on left and right side elements to the pivot element.

How does Quick Sort Partitioning Work

1. First find the "**pivot**" element in the array.
2. Start the left pointer at first element of the array.
3. Start the right pointer at last element of the array.

4. Compare the element pointing with left pointer and if it is less than the pivot element, then move the left pointer to the right (add 1 to the left index). Continue this until left side element is greater than or equal to the pivot element.
5. Compare the element pointing with right pointer and if it is greater than the pivot element, then move the right pointer to the left (subtract 1 to the right index). Continue this until right side element is less than or equal to the pivot element.
6. Check if left pointer is less than or equal to right pointer, then swap the elements in locations of these pointers.
7. Check if index of left pointer is greater than the index of the right pointer, then swap pivot element with right pointer.

Example:



Algorithm:

```

quickSort(array, lb, ub)
{
  if(lb < ub)
  {
    pivotIndex = partition(arr, lb, ub);
    quickSort(arr, lb, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, ub);
  }
}

```

RADIX SORT:

- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort.
- Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.
- During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.
- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant (LSD) to the most significant (MSD) digit. While sorting the numbers, we have **ten** buckets, each for one digit (0, 1, 2, ..., 9) and the number of passes will depend on the **length** of the number having maximum number of digits.

Example 1: Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

- In the first pass, the numbers are sorted according to the digit at ones place.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

- After this pass, the numbers are collected bucket by bucket. In the second pass, the numbers are sorted according to the digit at the tens place.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

- In the third pass, the numbers are sorted according to the digit at the hundreds place.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

- The numbers are collected bucket by bucket. After the third pass, the list can be given as final sorted list. 123, 345, 472, 555, 567, 654, 808, 911, 924.

Algorithm:

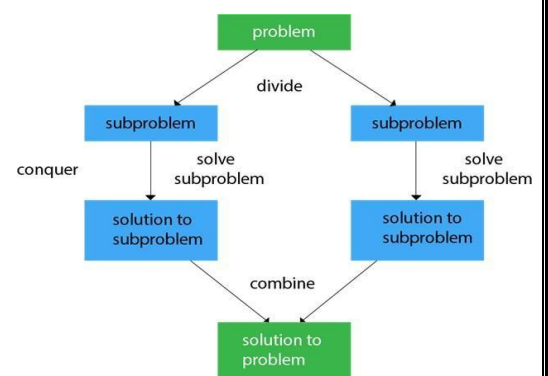
- Let **A** be a linear array of **n** elements **A[1], A[2], A[3].....A[n]**. Digit is the total number of digit in the largest element in array **A**.
- Input **n** number of elements in an array **A**.
- Find the total number of digits in the largest element in the array.
- Initialize **i=1** and repeat the steps 4 and 5 until(**i<=Digit**).
- Initialize the bucket **j=0** and repeat the steps 5 until(**j<n**).
- Compare the **ith** position of each element of the array with bucket number and place it in the corresponding bucket.
- Read the elements (**S**) of the bucket from **0th** bucket to **9th** bucket and from the first position to the higher one to generate new array **A**.
- Display the sorted array **A**.
- Exit.

Divide and Conquer:

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

- Divide and Conquer algorithm consists of a dispute using the following three steps.

- Divide** the original problem into a set of sub-problems.
- Conquer:** Solve every sub-problem individually, recursively.
- Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.



MERGE SORT:

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

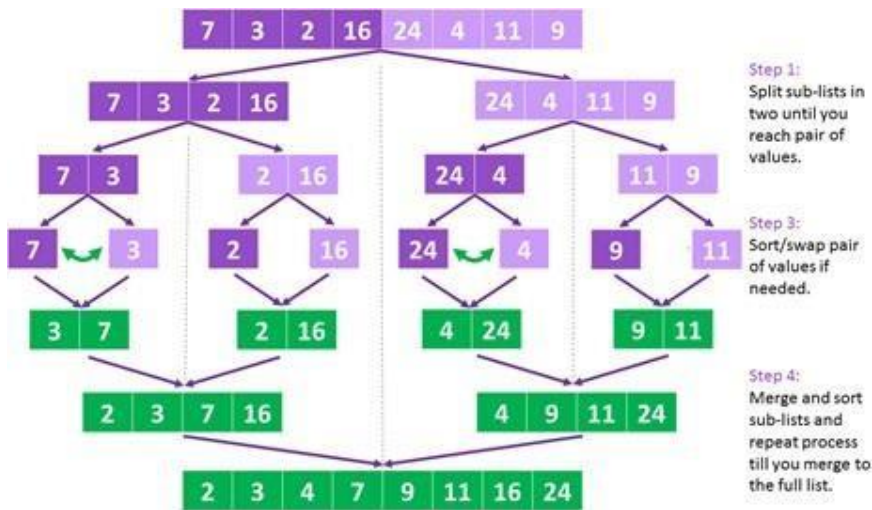
Implementation Recursive Merge Sort:

- The merge sort starts at the Top and proceeds downwards, “split the array into two, make a recursive call, and merge the results.”, until one gets to the bottom of the array-tree.

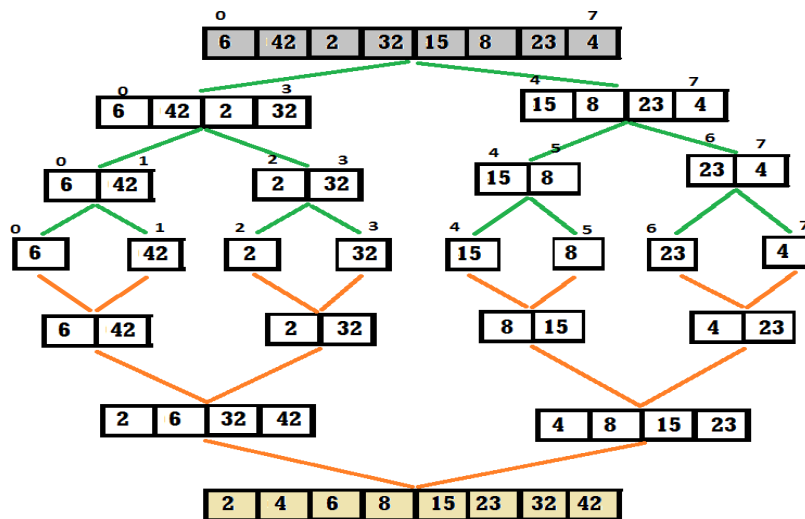
Example: Let us consider an example to understand the approach better.

- Divide the unsorted list into n sub-lists based on mid value, each array consisting 1 element
- Repeatedly merge sub-lists to produce newly sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list

Recursive Merge Sort Example:



Example 2:

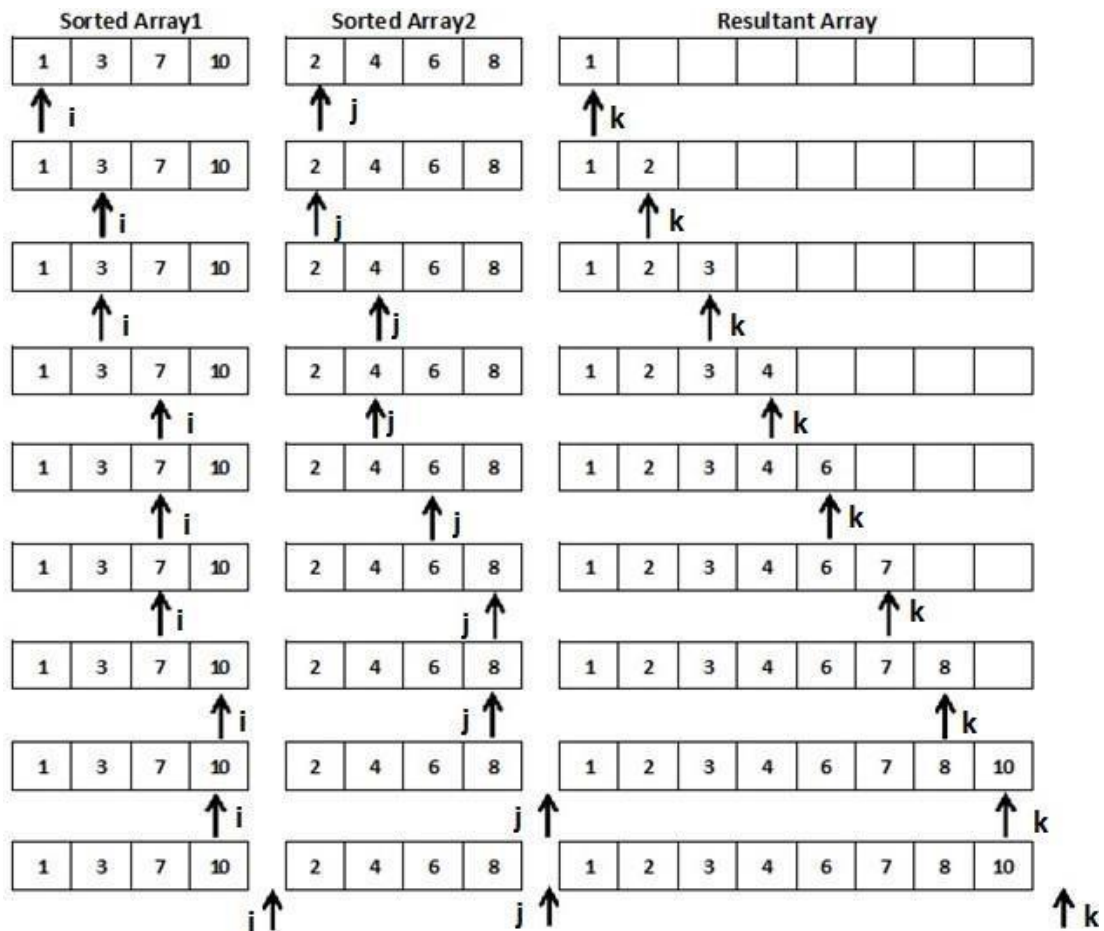


MergeSort Algorithm:

```

MergeSort(A, lb, ub)
{
    If lb < ub
    {
        mid = floor((lb+ub)/2);
        mergeSort(A, lb, mid)
        mergeSort(A, mid+1, ub)
        merge(A, lb, ub, mid)
    }
}
    
```

Two- Way Merge Sort:



Merge Algorithm:

Step 1: set $i, j, k=0$

Step 2: if $A[i] < B[j]$ then

copy $A[i]$ to $C[k]$ and increment i and k

else

copy $B[j]$ to $C[k]$ and increment j and k

Step 3: copy remaining elements of either A or B into Array C.

Time Complexities All the Searching & Sorting Techniques:

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

UNIT - II

LINKED LISTS

Syllabus :-

Linked Lists : Introduction, Representation of linked list in memory, single linked list, operations on single linked list, Reversing a single linked list, Application of single linked list to represent polynomial expression and sparse matrix manipulation, Advantages and Disadvantages of single linked list, circular linked list, Double linked list.

Assignment Questions :-

- 1) What is single linked list, write an algorithm to insert and delete a node in single linked list.?
- 2) What are the advantages & Disadvantages of single linked list?
- 3) What are the differences between arrays & linked lists?
- 4) Write an algorithm for reversing single linked list elements?
- 5) Explain applications of single linked list?
- 6) What is double linked list? write an algorithm for insert, delete and display the nodes in list?
- 7) What is circular linked list? Explain its operation?

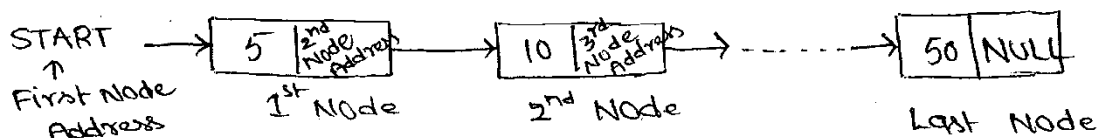
Linked list :-

Linked list is a linear collection of data elements. These elements are called nodes. For each node we are having two fields

- i) Data Field
- ii) Address Field.

→ Data field used to store the element (information)
→ Address field used to store the address of next node (element). So it is a pointer to store address.

- The last node not having next node, so the address field of last node is NULL.
- START-pointer, it stores the first node address in the list.
- We can traverse entire list by using START. To find the second node address we have the address in the first node.



- Using this technique the individual nodes of a list will form chain of nodes.
- If $START = NULL$, then the list is empty list.
- In order to form a linked list, we need a structure called node. which has two fields data and next.

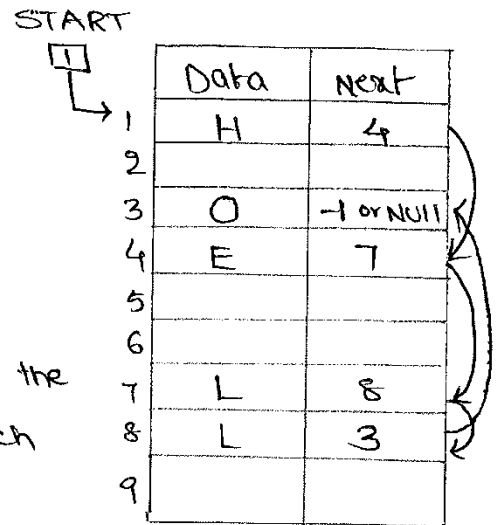
data → stores the information part
Next → Address of the next node.

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

Memory representation of Linked list :-

(2)

- START is used to store the address of the first node.
- In this example, start = 1, so the first node stores at address 1, which element is H.
- The corresponding Next stores the address of the next node, which is 4.
- So, we will look at address 4 to fetch the next data item.
- The second data element is obtained from address 4 is E, again we see the corresponding Next to go to the next node.



- We repeat this procedure until we reach a position where the Next field contains -1 or NULL, then we denote last element in the list.
- Remember that the nodes of a linked list need not be consecutive memory locations, here we are storing the elements at 1, 4, 7, 8, 3.

Memory Allocation & De-allocation :-

- If we want to add new node into already existing list in the memory, we first find free space in memory then store the information.
- Computer will maintain a list of all free memory cells. The list of available space is called freepool.
- For pointing free space in memory we have a pointer called AVAIL. It stores the address of first free space in memory.

- After inserting new node into the list, the next available free space is pointing the AVAIL.

START	Data	Next
1	H	4
2		5
3		-1
4	E	6
5		8
6	L	7
7	L	9
8		3
9	O	-1

- Deleting a node from the list, the space occupied by node is given back to free pool so the memory can reuse.

- collecting all remaining space into free pools, this process called Garbage collection.

Dynamic Memory :-

- We are having 4 memory management functions, called `calloc()`, `malloc()`, `realloc()`, and `free()`
- All the functions are available in "stdlib.h"

1. `malloc()` :- Allocate required size of bytes and returns a pointer first byte of allocated space.

Variable = (Datatype *) malloc (sizeof (datatype));

Ex:- Ptr = (int *) malloc (sizeof (int));

2. `Calloc()` :- Allocate space for array elements, initialize to zero and return a pointer to memory

Ex:- variable = (Datatype *) calloc (N, sizeof (datatype));

Ptr = (int *) calloc (20, sizeof (int));

3. `realloc()` :- change the size of previously allocated space

variable = realloc (variable, new size);

4. `free()` :- Deallocate the previously allocated memory space

free (variable);

Ex:- free (Ptr);

Arrays vs Linked list :-

(3)

1. Both are linear collection of data elements
2. → Arrays will allocate the memory in sequential order.
→ Linked list will allocate the memory for elements in random
3. → In Arrays insertion/deletion is very difficult because if you delete first element, shifting all elements to previous locations
→ In Linked list insertion/deletion can perform at any point, just by changing the next field of a node we can perform operations.
4. → In arrays we can add fixed size no. of elements
→ In linked list we can add any number of elements.
5. → In arrays memory allocation at compile time, some times memory space wastage.
→ In linked list memory allocation at Run-Time, by using dynamic memory allocation functions we can perform.

Single linked list :-

- Single linked list is the simple type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- Traversal of linked list is only one way, from START to end of Node.
- Operations of single linked list
 1. Traversing
 2. Searching
 3. Insertion
 4. Deletion.

1. Traversing :-

- Traversing of linked list means accessing the nodes of the list in order to perform some operations.
- Linked list contains the pointer START, which stores the address of the first node in the list.
- For the last node the next field address is NULL.
- We are taking one pointer PTR for accessing the nodes.

Algorithm Traversal ()

```
{  
  [Initialize] set PTR := START;  
  Repeat steps while PTR != NULL  
    Apply process PTR → data ;  
    Set PTR = PTR → next ;  
  End loop ;  
}
```

- For counting number of nodes in a list

Algorithm countnodes ()

```
{  
  [Initialize] set count := 0 ;  
  [Initialize] set PTR := START ;  
  Repeat steps while PTR != NULL  
    set count := count + 1 ;  
    set PTR := PTR → next ;  
  End loop  
  write count ;  
}
```

2. Searching :-

(4)

- Searching a list means to find particular element presented in the linked list or not
- There are two outcomes for searching, one is node address and other is NULL
- The given key element is presented in the list then it will return the node address, if it is not presented then it will return NULL.

Algorithm Search (item)

```
{
  [Initialize] Set POS := NULL;
  [Initialize] Set PTR := START;
  Repeat while PTR != NULL
    if item == PTR → data then
      Set POS := PTR;
    else
      Set PTR := PTR → next;
  return (POS);
}
```

3. Insertion :-

- If the list is already containing the nodes then we can insert a new node in following ways
 - 1) At beginning of the list
 - 2) At End of the list
 - 3) At particular position in the list.
- If START = NULL then the list is empty
- If AVAIL = NULL then no free memory cells in the system.

(i) Beginning of the list :-

- For inserting new node in to the list first check memory space is available or not
- If the memory is available (AVAIL != NULL) then create new node and AVAIL pointing to the next free space
- Now, insert the values for node. Directly insert data item into data field and in the Next field we can store the first node address
- Now, New node is the first node in the list so store the new node address into START.
- Finally, we are inserted new node at beginning of list.

Algorithm insert_beg (item)

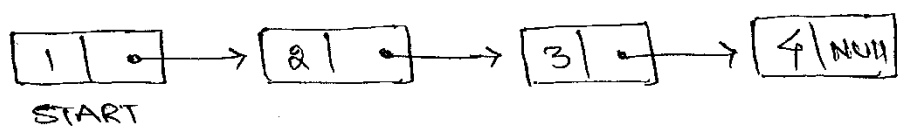
```

{
    if AVAIL := NULL
        Write "No memory for creation";
        Go to Exit;
    Set New_node := AVAIL;
    Set AVAIL := AVAIL -> next;
    Set New_node -> data := item;
    Set New_node -> next := START;
    Set START := New_node;
}
    
```

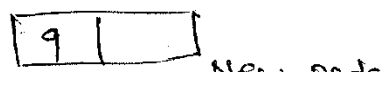
New node creation
 values insertion into New_node

Example :-

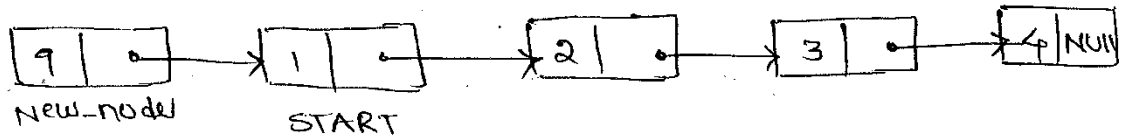
- Add new node containing data 9 into list.



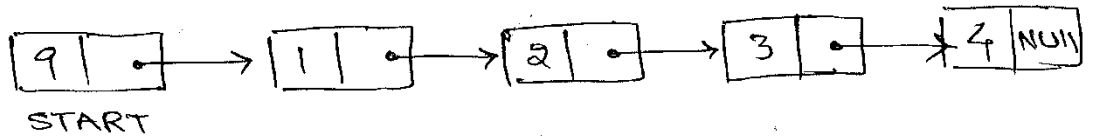
- Allocate memory for new_node containing data 9



- Add new_node at beginning of the list by taking ⁽⁵⁾ the next part of new_node containing address of START



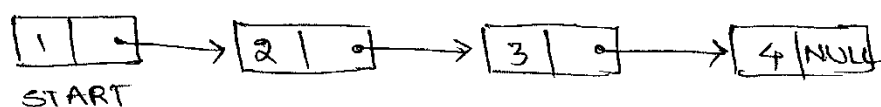
- Now make START to point to the first node of list.



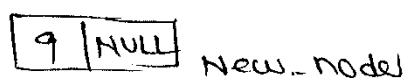
(ii) Insert at End of list :-

- Initially we are taking one pointer PTR for accessing the nodes in the list and it is initialized with START.
- check free memory space is available for new_node
- insert the field value of new_node, data part we can directly insert item and next part is NULL because the new_node is the last node in the list.
- Now, we are moving to the last node in the list by using PTR
- Add new node address to the next field of the last node then we are creating link between new node and previous last node
- Finally we are inserted new node at the end of list.

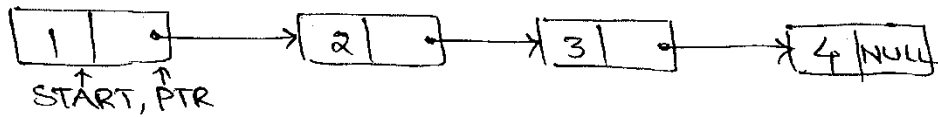
Example :- Add new node containing '9' at end of list.



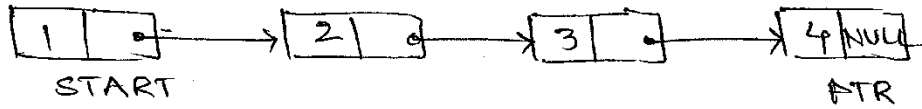
- Allocate memory for new_node containing data 9 and next address NULL



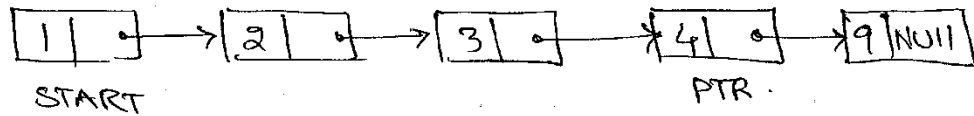
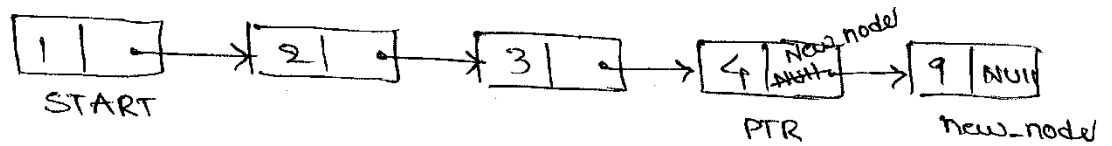
- Take a pointer variable PTR with value of START.



- Move PTR to end of list



- Add new nodes at end of the list and change the next field of PTR nodes by assigning new nodes address



Algorithm insert_end (item)

{

if AVAIL = NULL

write "No memory for new node";

Go to Exit;

Set New_node := AVAIL;

Set AVAIL := AVAIL → next;

Set New_node → data := item;

Set New_node → next := NULL;

Set PTR := START;

Repeat steps while PTR → next != NULL do

Set PTR := PTR → next;

End loop

Set PTR → next := New_node;

}

(iii) Insert at particular position :-

6

- Inserting new-node into the list first check memory is available or not, if it is available create new-node.
- Initially taking one pointer PTR assigned by START, then it is pointing to the first node.
- Now we are moving from one node to other, upto given POS value.
- Now we are inserting the field value of new-node, data part by item and next field by PTR → next.
- And now change the PTR next field address by new-node.
- Finally we are inserting new-node at given particular POS location.

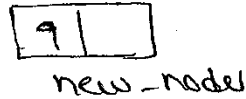
Algorithm insert_pos(POS, item)

```
{
    if AVAIL = NULL
        Write "No memory for new-node";
        Goto Exit;
    new_node := AVAIL;
    AVAIL := AVAIL → next;
    PTR := START;
    i := 1;
    while i < POS - 1
        PTR = PTR → next;
        i++;
    new_node → data := item;
    new_node → next := PTR → next;
    PTR → next := new_node;
}
```

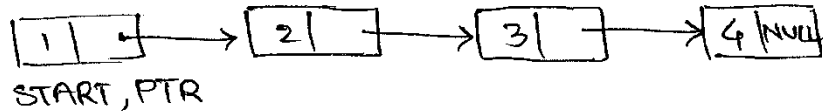
Example 3 - Add new node containing '9' at particular position.



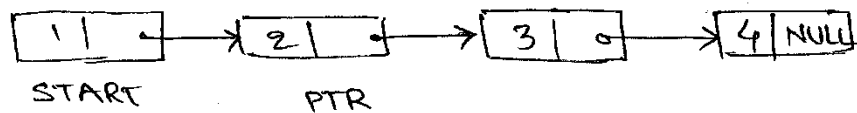
- Allocate memory for new_node containing data 9.



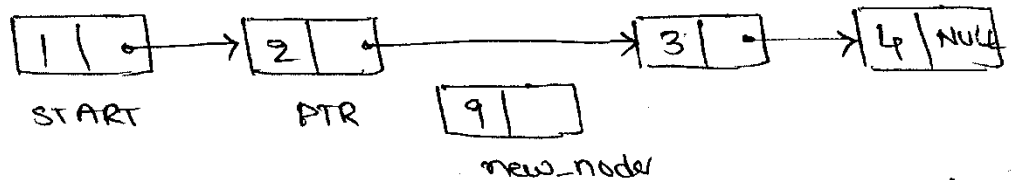
- Take a pointer variable PTR with value of START



- Move PTR to given POS = 3 \Rightarrow pos-1 position

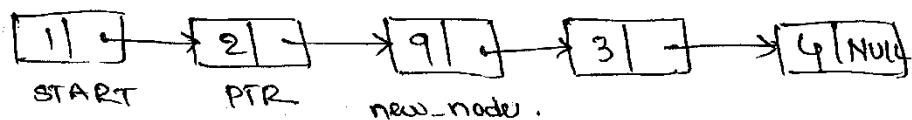
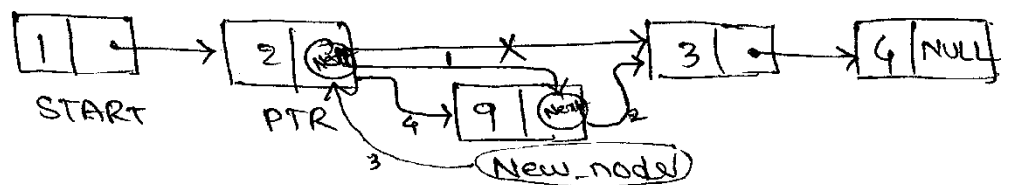


- Add new_node into the list after PTR node.



- Now copy the PTR \rightarrow Next address and assigned into new_node \rightarrow next, we are creating link between new_node and next node containing data 3

- And change PTR next address, it points to new_node then finally we are inserting new_node at given position.



4. Deletion :-

(7)

- If we are already containing more than one node in the list, then we can perform different type of deletion operations on list

(i) At Beginning of the list

(ii) At End of the list

(iii) At particular position of a node in list

(i) At Beginning of the list :-

- First, we need to check whether the list is having the nodes or not, if nodes are presented in the list then only we can delete the node from the list otherwise not possible.

- We are taking a pointer variable PTR assigned by START.

- Now, we are changing the START position, because after deleting first node from the list second node is the starting node in the list

- Now, we can delete the PTR node from the list

- Finally we are deleting beginning node from the list

Algorithm del_beg ()

{

if START = NULL

write "No nodes in the list";

goto Exit;

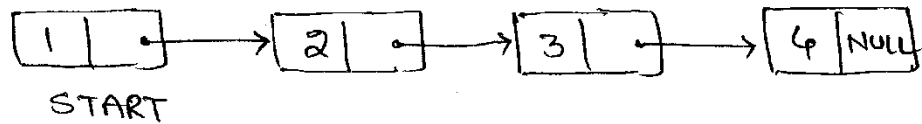
PTR := START;

START := START → Next;

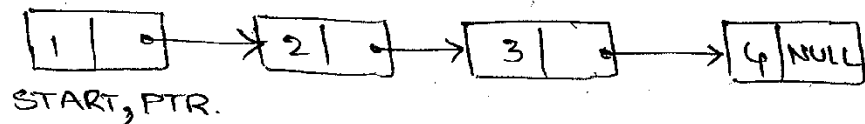
free (PTR);

}

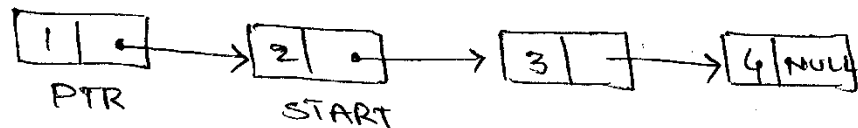
Example :- Deleting first node from the list containing data '1'.



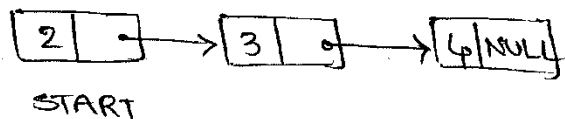
- Take one PTR variable containing START address, then it is pointing the first node.



- Now, change the START position to the next node by START \rightarrow next, then it is pointing second node



- Now, delete the PTR node from the list then the final list is



- Finally we are deleting beginning node from the given list.

(ii) Deleting node at End of list :-

- First, we need to check whether the list is having the nodes or not, if nodes are presented in the list then only we can delete the node from the list otherwise it is not possible.

- Now, we are taking a PTR variable assuming START address

- Now move PTR, pointing the last node in the list meanwhile we are taking PREPTR pointing the previous of last nodes.

- Now change the address of PREPTR to NULL

- Delete the PTR node from the list.

- Finally we are deleting the last node from list. (8)

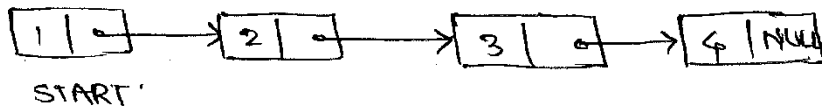
Algorithm del_end ()

```

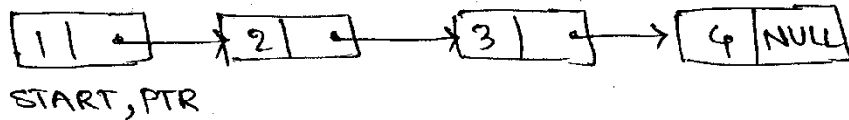
{
  if START = NULL
    write "No nodes in the list";
    goto Exit;
  PTR := START
  while PTR → Next ≠ NULL do
    PREPTR := PTR;
    PTR := PTR → Next;
  PREPTR → Next := NULL;
  free ( PTR );
}

```

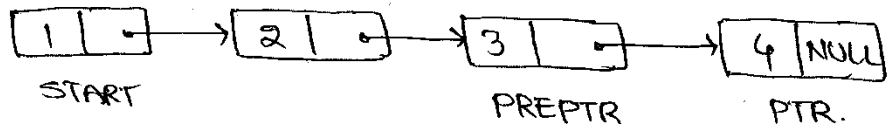
Example :- Delete the last node from list containing data '4'



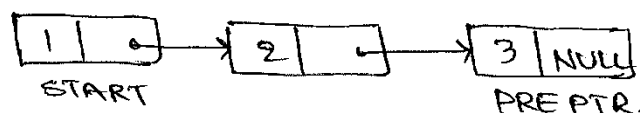
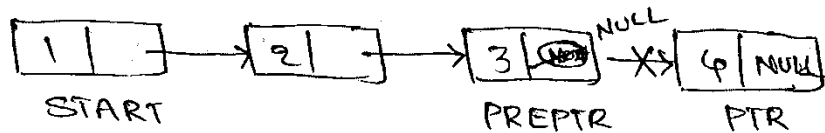
- Take a pointer variable PTR and it points to first Node.



- Move PTR from starting to end of the list meanwhile take PREPTR pointing to the previous of PTR node.



- Now change the next value of PREPTR then we are breaking link between PREPTR and PTR and delete the PTR node from list



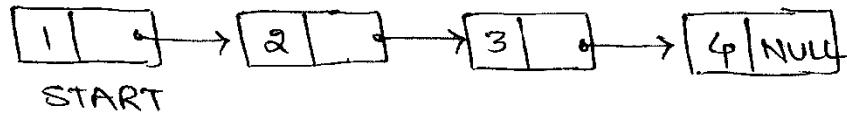
(iii) Deleting particular nodes from list :-

- First check the list containing nodes or not, if the nodes are present then only we can delete the node from the list otherwise it is not possible.
- Initially take PTR variable containing START address.
- Now move PTR upto given POS value meanwhile take PREPTR, pointing previous of PTR node.
- Now change the address of PREPTR \rightarrow Next by assigning a value of PTR \rightarrow Next, now we are creating link between PREPTR and next nodes of PTR.
- Now we can delete PTR node from the list, finally we are deleting particular nodes from the list.

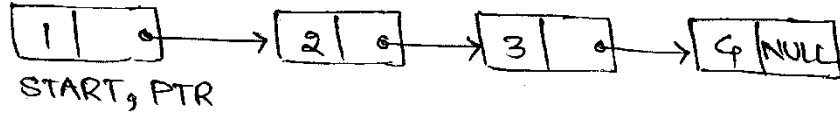
Algorithm del-POS (POS)

```
{
  if START = NULL
    Write "NO nodes in the list";
    Goto Exit;
  PTR := START;
  i := 1;
  while i < POS do
    PREPTR := PTR;
    PTR := PTR  $\rightarrow$  Next;
  PREPTR  $\rightarrow$  Next := PTR  $\rightarrow$  Next;
  free (PTR);
}
```

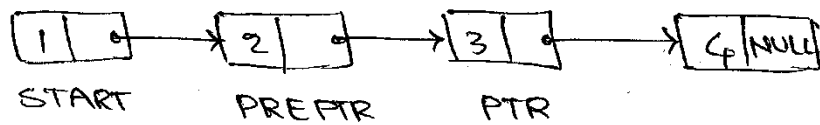

Example :- Delete a node at POS = 3 from list. ①



- Take PTR variable pointing to first node.



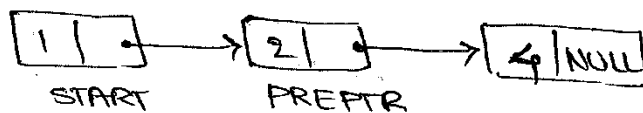
- Move PTR to POS = 3 location meanwhile take PREPTR pointing previous of PTR node.



- Change the PREPTR next field address by PTR next field address then we are pointing or linking PREPTR node and next nodes of PTR.



- Now delete PTR from the list then the final list as follows.



- Finally we are deleting particular node from the list.

Reversing A single linked list :-

- First create a single linked list having nodes.

- taking two pointer variables PTR1 = NULL and PTR2

- Repeat the following process upto the last node

1). Take first two nodes from the list and PTR2 is pointing to the Second Node.

2) change the first node next field address assign the PTR1 value.

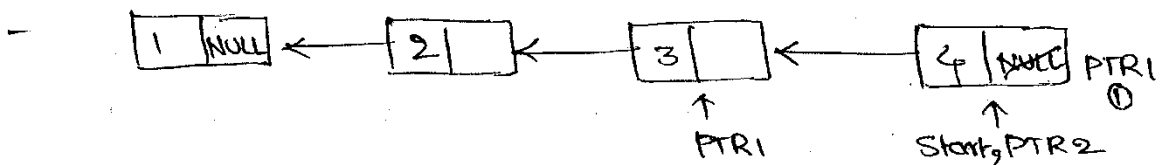
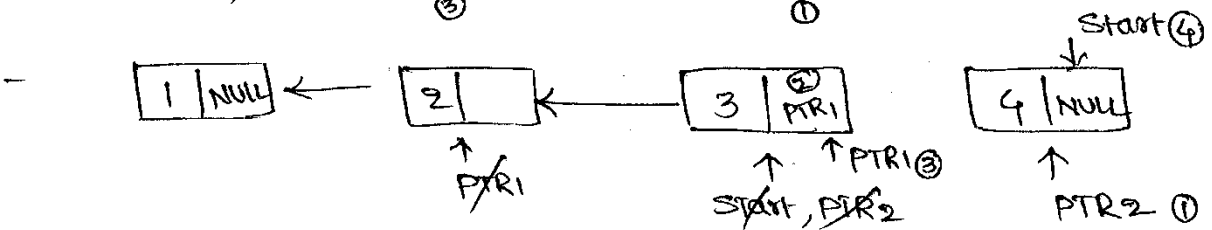
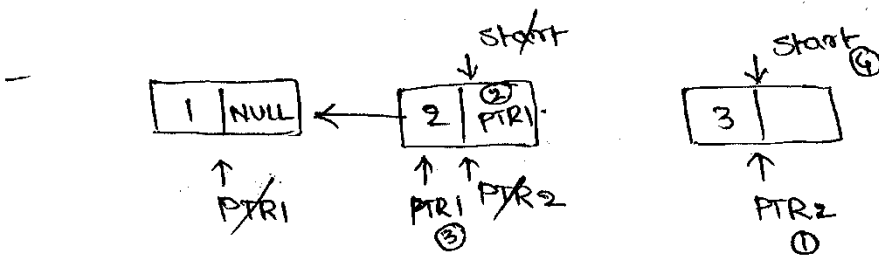
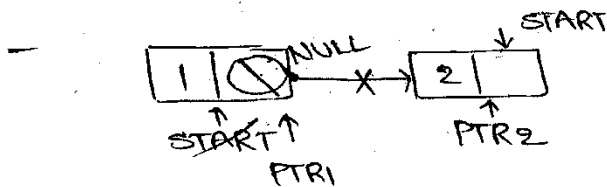
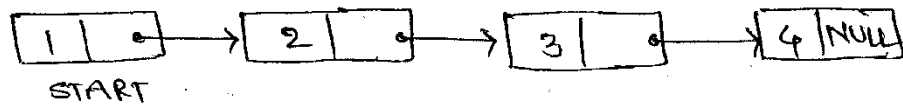
3) Now pointing the first node as PTR1

4) change the START, pointing to PTR2 because PTR2 is the starting node if the list is in reverse order.

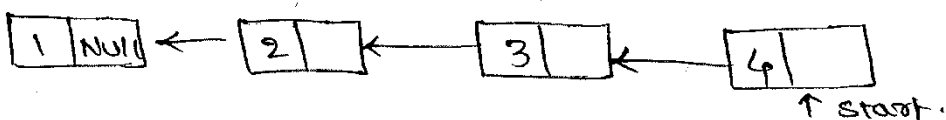
- After getting or moving to the last node change the next field of first node after reversing by PTR1

- Finally we are reversing the given single linked list.

Example :-



- Finally the reverse list is



Algorithm :-

(10)

Algorithm reverse ()

{

PTR1 := NULL, PTR2;

While START do

{

PTR2 := START → Next;

START → Next := PTR1;

PTR1 := START;

START := PTR2;

}

START → Next := PTR1;

}

Advantages & Disadvantages of Single linked list :-

Advantages :-

- 1) Insertion and Deletion can be done easily.
- 2) It doesn't need movement of elements for insertion and deletion
- 3) Size is not fixed so there is no space wastage
- 4) We can increase the size of the list according to our requirement.
- 5) Elements may or may not be stored in consecutive memory locations, even though we can store the data in computer
- 6) It is less expensive.

Disadvantages:-

- 1) It requires more space because pointers are also stored the information.
- 2) Different amount of time is required to access the elements in the list.
- 3) We can't traverse from last, only traverse from beginning.
- 4) It is not easy to sort the elements stored in the linear linked list.

Applications of linked list :-

- We are having two types of applications

- 1) Polynomial Representation
- 2) Sparse Matrix Manipulation.

1) Polynomial Representation :-

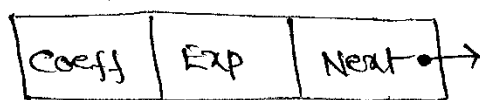
- Polynomials are the expressions containing number of terms with non-zero coefficients and exponents

$$P(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_{n-1}x^{n-1} + a_nx^n$$

where a_i is non-zero coefficient
 n is non-negative integer.

- In the linked representation of Polynomials, each term is considered as node and the node containing 3 fields

- 1) coefficient field
- 2) Exponent field
- 3) Next node address field



- The coefficient field holds the value of coefficient⁽¹¹⁾ of a term and the exponent field contains the exponent value of the term.
- Next field contains the address of the next term in the polynomial.
- In C, the structure of polynomial nodes is

```

struct Polynode
{
    int coeff;
    int exp;
    struct Polynode *Next;
};

```

- Algorithm for creation of polynomial equation in linked list

Algorithm create_poly ()

```

{
    Read c, e;
    while coeff != 0 do
    {
        if START == NULL then
        {
            new_node := AVAIL;
            AVAIL := AVAIL → Next;
            new_node → coeff := c;
            new_node → exp := e;
            new_node → Next := NULL;
        }
        else
        {
            PTR := START;
            while PTR → Next != NULL do
                PTR := PTR → Next;
            new_node := AVAIL;
            AVAIL := AVAIL → Next;
        }
    }
}

```

```

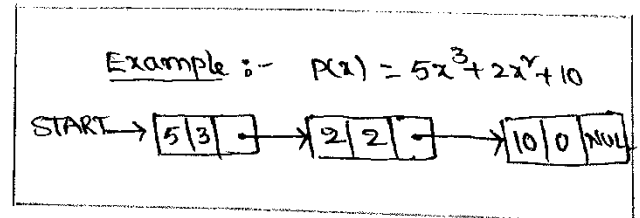
new_node → coeff := C ;
new_node → exp := e ;
new_node → Next := NULL ;
PTR → Next := new_node ;

```

```

}
Write "Enter the coefficient and Exponent value";
Read c, e ;
} // closing of while
} // closer of function.

```



Operations of Polynomial :-

- We are having 4 types of Polynomial operations, those are

- 1) Evaluating Polynomial at given value
- 2) Addition of two polynomials
- 3) Subtraction of two polynomials
- 4) Multiply two polynomials

Addition of two Polynomials :-

- Initially take two polynomials P & Q and Resultant R
- We have to compare their starting terms from first node and moving towards end one by one
- Three PTR variables to represent their lists P PTR, Q PTR and R PTR respective P, Q and R.
- there are 3 cases during the comparison between the terms of polynomials
 - 1) The exponent of two terms are equal, the coefficients of two terms are added and a new term is created with the value

$RPTR \rightarrow \text{coeff} := PPTR \rightarrow \text{coeff} + QPTR \rightarrow \text{coeff}$ (12)

and

$RPTR \rightarrow \text{Exp} := PPTR \rightarrow \text{Exp};$

2) If the exponent of P is greater than the exponent of Q then the duplicate of current term P is created and inserted in polynomial R

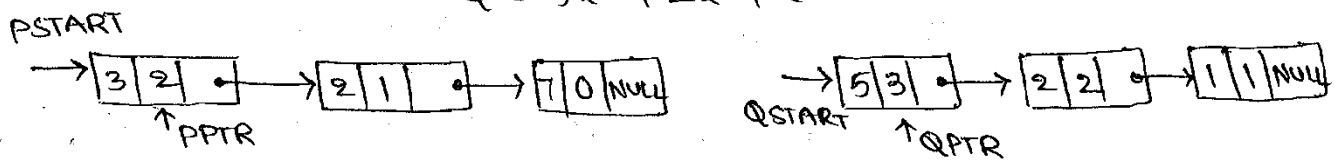
3) If the exponent of P is smaller than the exponent of Q then the duplicate of current term Q is created and inserted in polynomial R

- Append the remaining terms in the either polynomials to the resultant polynomial R.

Example :- Take two polynomials P, Q

$$\text{Let } P = 3x^2 + 2x + 7$$

$$Q = 5x^3 + 2x^2 + x$$

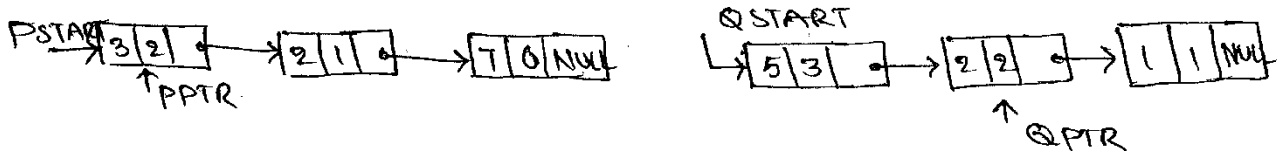
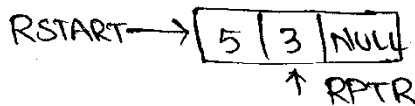


- Compare the exponents of P and Q

$$\text{Exp}(P) < \text{Exp}(Q) \Rightarrow PPTR \rightarrow \text{Exp} < QPTR \rightarrow \text{Exp}$$

$$2 < 3$$

- Add the Q term into the resultant polynomial R and move to next node.

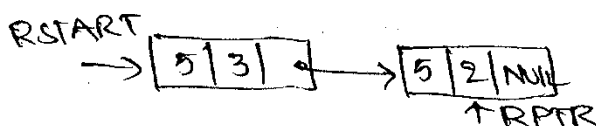


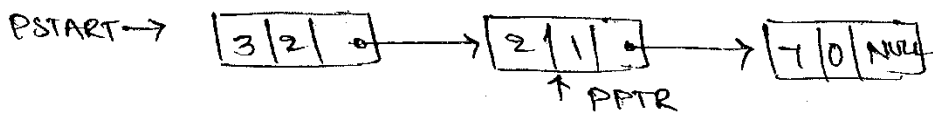
- Compare the exponents of P and Q

$$\text{Exp}(P) = \text{Exp}(Q) \Rightarrow 2 = 2 \text{ then}$$

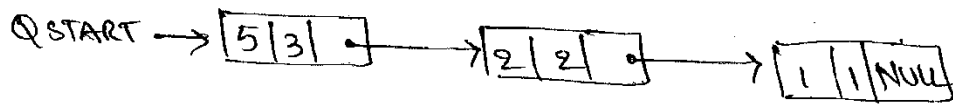
$$\text{Add two coefficients } 3 + 2 = 5$$

- Now the resultant polynomial R is





↑ PPTR



↑ QPTR

- Now compare the exponents of two polynomials

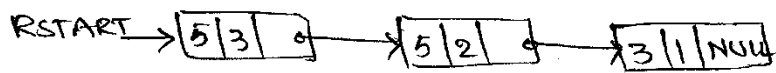
$$\text{exp}(P) = \text{exp}(Q) \Rightarrow 1 = 1 \text{ then}$$

add the two coefficients $2+1=3$

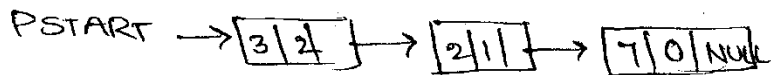
- Now the new node

3	1	•
---	---	---

 added to resultant R and move PPTR and QPTR to the next nodes



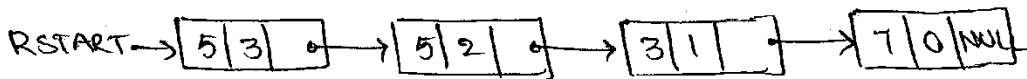
↑ RPTR



↑ PPTR

QPTR = NULL

So, directly append the P remaining terms into the resultant R polynomial.



↑ RPTR

- Finally the resultant polynomial R is

$$R(x) = 5x^3 + 5x^2 + 3x + 7$$

- Algorithm for addition of two polynomials

Algorithm Add-Poly ()

{

PPTR := PSTART, QPTR := QSTART, RPTR := RSTART;

while PPTR != NULL and QPTR != NULL do

{

if PPTR → EXP = QPTR → EXP then

{

new_node := AVAIL;

AVAIL := AVAIL → next;

.....

RPTR := new_node;

RPTR → coeff := PPTR → coeff + QPTR → coeff ;

RPTR → Exp := PPTR → Exp ;

RPTR → Next := NULL ;

PPTR := PPTR → Next ;

QPTR := QPTR → Next ;

}

if PPTR → Exp > QPTR → Exp then

{

new_node := AVAIL ;

AVAIL := AVAIL → Next ;

RPTR := new_node ;

RPTR → coeff := PPTR → coeff ;

RPTR → Exp := PPTR → Exp ;

RPTR → Next := NULL ;

PPTR ⇒ PPTR → Next ;

}

if PPTR → Exp < QPTR → Exp then

{

new_node := AVAIL ;

AVAIL := AVAIL → Next ;

RPTR := new_node ;

RPTR → coeff := QPTR → coeff ;

RPTR → Exp := QPTR → Exp ;

RPTR → Next := NULL ;

QPTR := QPTR → Next ;

}

} // End of while loop

while PPTR != NULL do

{

new_node := AVAIL ;

AVAIL := AVAIL → Next ;

RPTR := new_node ;

RPTR → coeff := PPTR → coeff ;

RPTR → Exp := PPTR → Exp ;

RPTR → Next := NULL ;

PPTR := PPTR → Next ;

}

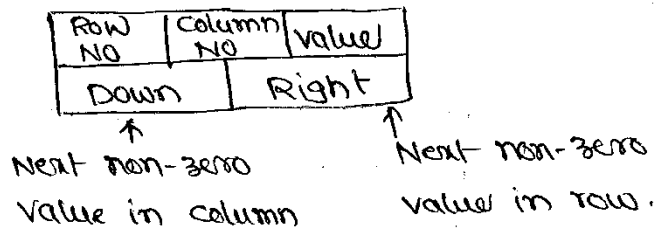
```

while QPTR != NULL do
{
    new_node := AVAIL;
    AVAIL := AVAIL → Next;
    RPTR := new_node;
    RPTR → coeff := QPTR → coeff;
    RPTR → Exp := QPTR → Exp;
    RPTR → Next := NULL;
    QPTR := QPTR → Next;
}
} // End of function.

```

(2) Sparse Matrix Manipulation :-

- Sparse matrices are those matrices which have majority of their elements equal to zero.
- The Node representation of sparse matrix is



- In C, the structure of sparse matrix is

```

struct sparse_node
{
    int row, column, value;
    struct sparse_node *down, *right;
}

```

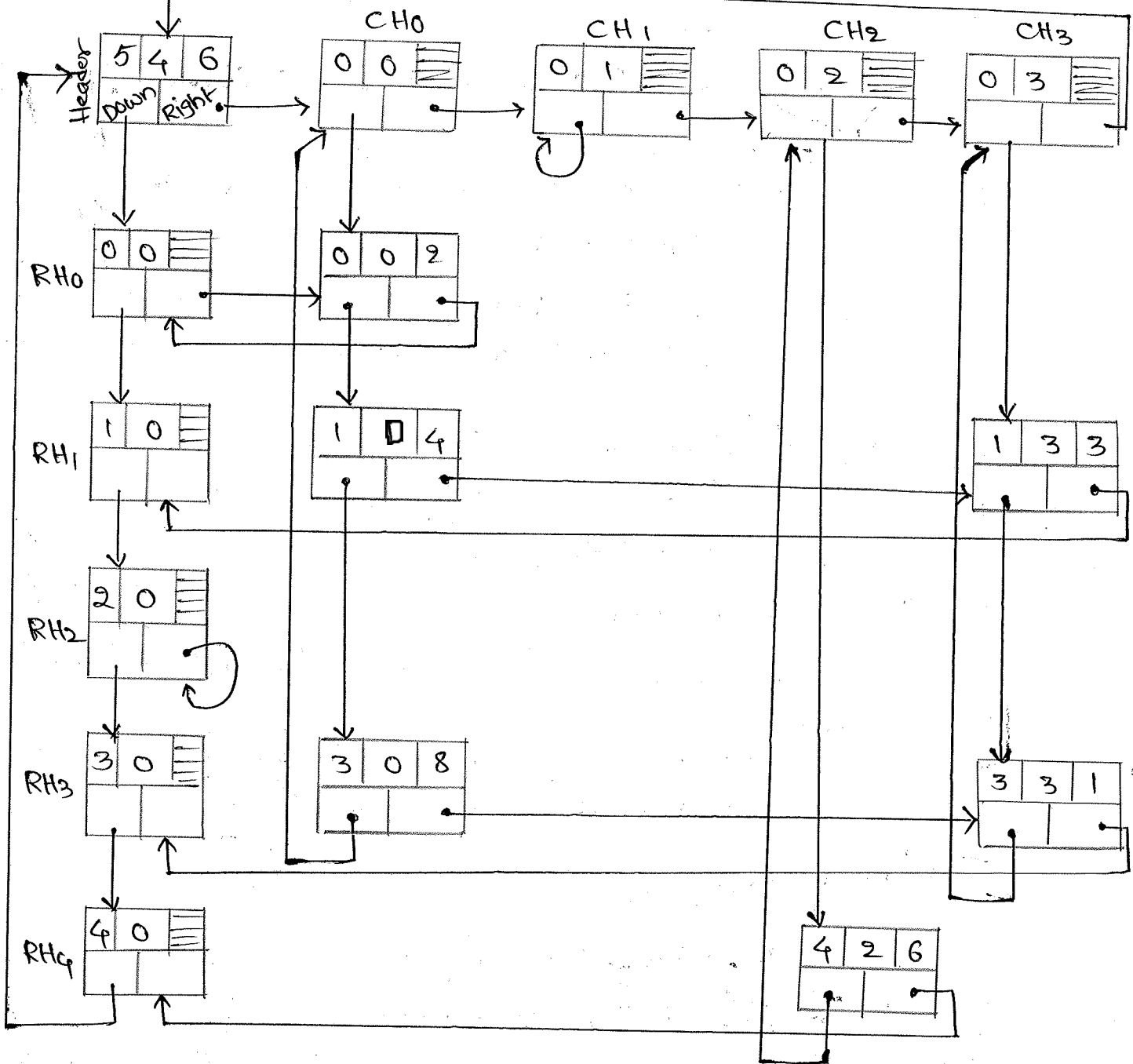
- In dummy header we are maintaining no. of rows and no. of columns and also maintain no. of non-zero elements in the matrix.

Example :- Let the matrix is

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

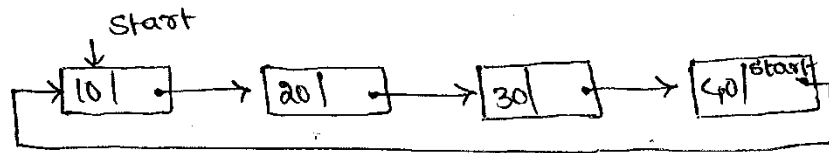
- Let the row headers are represented as RH_0, RH_1, RH_2, \dots and the column headers are represented as CH_0, CH_1, CH_2, \dots

- Now the representation of sparse matrix is



Circular linked List :-

- A linked list where the last node points the starting node is called the circular linked list.
- There is no beginning and ending of list.



- In C, the structure of circular linked list is

Struct node

```
{ int data;  
  struct node *Next;  
};
```

- The operations of circular linked list is

- 1) Insertion of a node
- 2) Deletion of a node.

1) Insertion of a node :-

- We can insert a new node in a circular linked list in 2 ways

- (i) Inserting a node at beginning
- (ii) Inserting a node at ending

(i) At beginning :-

Algorithm insert_beg (item)

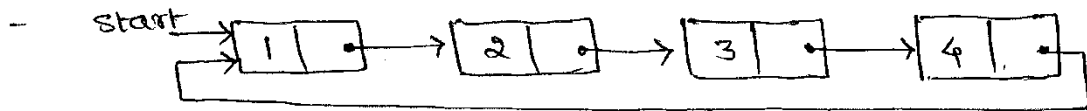
```
{  
  if AVAIL = NULL  
    Write "No memory for creating a node";  
    Goto Exit;  
  new_node := AVAIL;  
  AVAIL := AVAIL → Next;  
  new_node → data := item;  
  new_node → Next := start;  
  Ptr := start;  
  while Ptr → Next != start do  
    set Ptr := Ptr → Next;
```

ptr → Next := new_node;

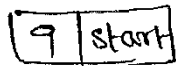
start := new_node;

}

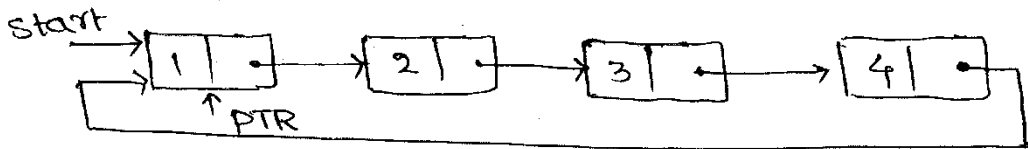
Example:- Inserting a new node of data '9' into circular linked list



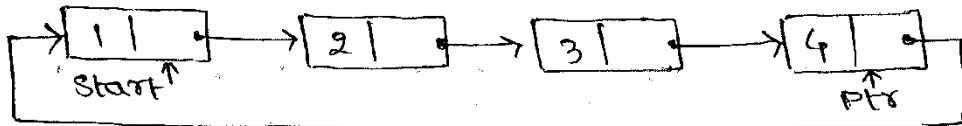
- Allocate memory for the new node and initialize its Data part 9 and Next by start



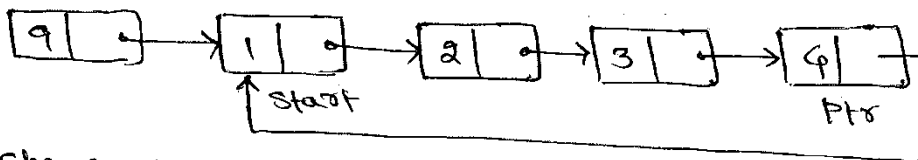
- Take a pointer variable PTR that points to START node



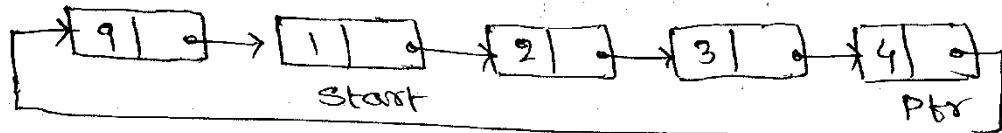
- move PTR so that it points to last node



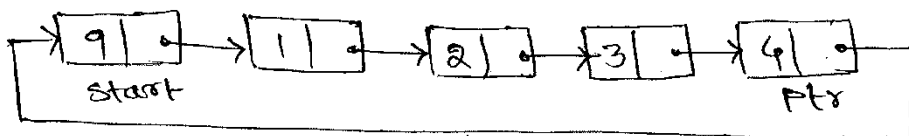
- Add the new node at beginning or starting of list



- Now change the last node next address to the new_node



- change the start to the new node



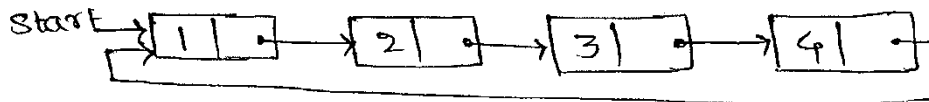
- Finally we are inserted new node at beginning of the circular linked list.

(ii) At Ending :-

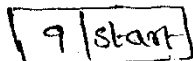
Algorithm insert_end (item)

```
{ if AVAIL = NULL then
    Write "No memory for creating new_node";
    Goto Exit;
new_node := AVAIL;
AVAIL := AVAIL → Next;
new_node → data := item;
new_node → Next := Start;
Ptr := Start;
While Ptr → Next != Start do
    Ptr := Ptr → Next;
Set Ptr → Next := new_node;
}
```

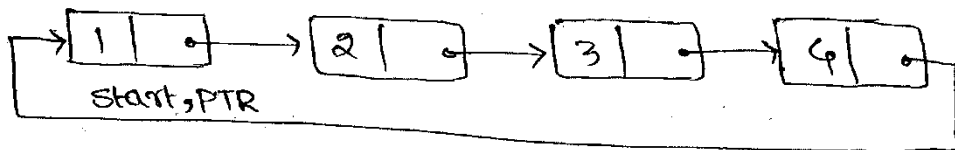
Example :- Insert new node into list having data 9



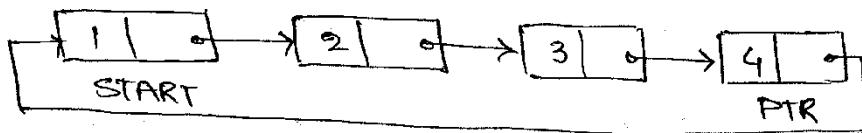
- Allocate new memory for new node and data 9, next start



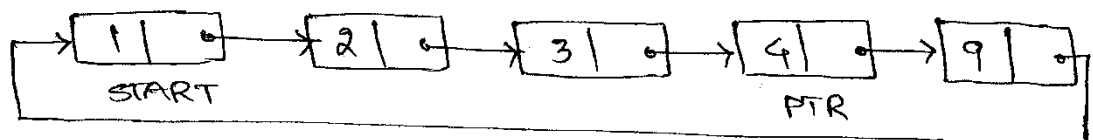
- Take a pointer variable PTR which initially points to START



- move PTR upto last node, PTR points to last node



- Add new node after the PTR node



- Finally we are inserted new node at end of the circular linked list.

(2) Deletion of a node :-

(16)

- We can delete the node from circular linked list in two ways

- (i) Deleting a node at beginning
- (ii) Deleting a node at ending.

(i) At beginning :-

Algorithm: Delete_beg ()

{ if START = NULL

Write "No nodes in the list";

Goto Exit;

Set PTR := START;

While PTR → Next ≠ START do

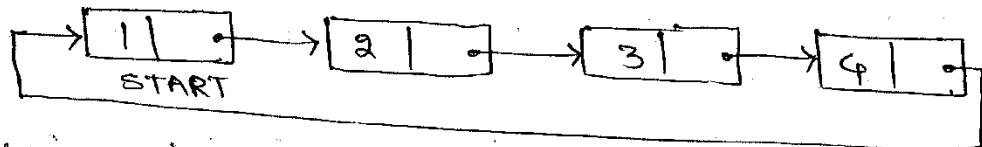
PTR := PTR → Next;

PTR → next := Start → Next;

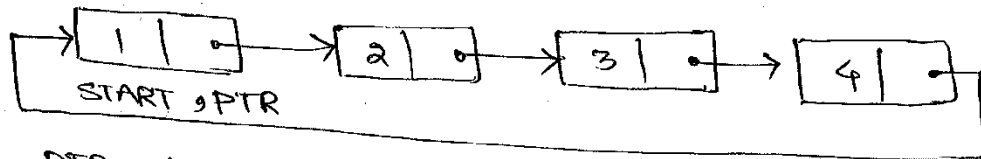
free (START);

} START := PTR → Next;

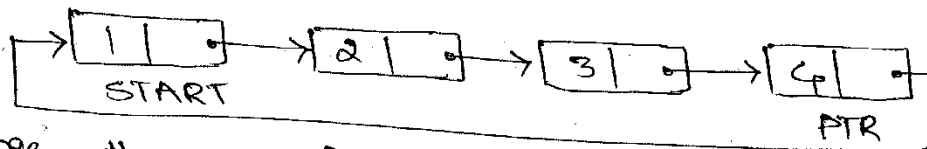
Example :- Deleting first node from the list



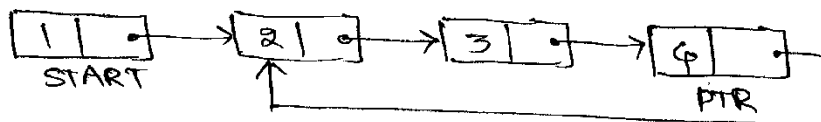
- Take a pointer PTR which points to the first node in list



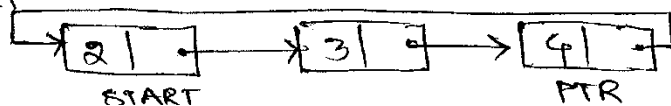
- Move PTR to last node in the list



- Change the next field of the PTR node to second node in list



- Delete the START node from list and change to next node as START

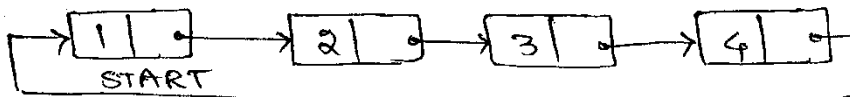


(iii) At Ending :-

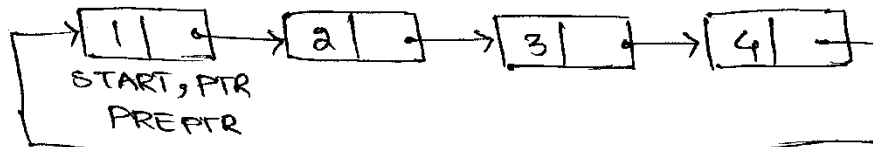
Algorithm Delete_end ()

```
{ if START = NULL
    write "No nodes in the list";
    Go to Exit;
PTR := START;
while PTR → Next != START do
    PREPTR := PTR
    PTR := PTR → Next;
    PREPTR → Next := START;
    free ( PTR );
}
```

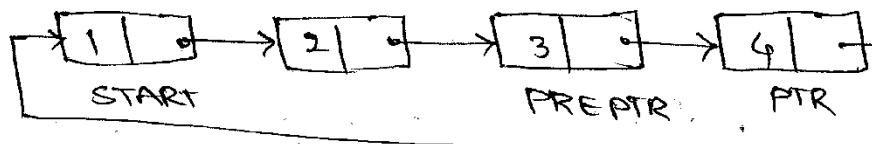
Example:- Deleting last node from the list.



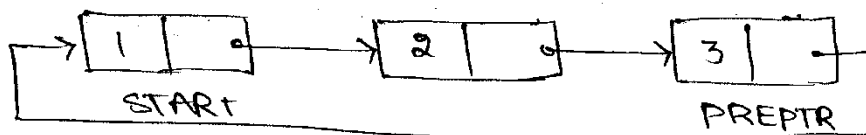
- Take two pointer variables PREPTR and PTR, initially point START



- move PTR to last node and PREPTR is points to previous of PTR node



- change the next field of PREPTR node and delete the PTR node from the list



- Finally we are deleting the last from the given circular linked list.

Double linked list :-

(17)

- A double linked list is more complex type of linked list which contains a pointer to the Next as well as Previous nodes in the sequence.
- In double linked list we can access both the successor node (Next node) and predecessor node (Previous node) for any arbitrary node in the list.
- Each node in double linked list



- In C, the structure of Double linked list is

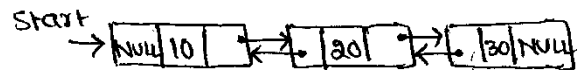
Struct node

{ int data;

struct node *prev;

struct node *next;

};



- If you are having single node in the list then prev and next values are null.
- For the first node always prev is NULL and for the last node always next is NULL.
- There are 2 operations on double linked list, those are
 - 1) Insertion
 - 2) Deletion.

1) Insertion :-

- We can insert new node in double linked list in 3 ways
 - (i) Inserting new node at beginning
 - (ii) Inserting new node at ending
 - (iii) Inserting new node at position given.

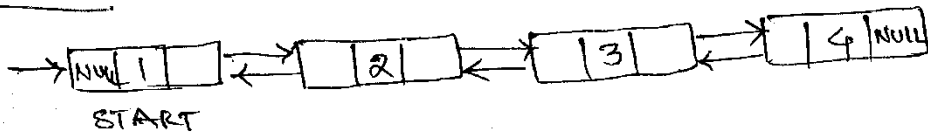
(i) At Beginning :- Algorithm insert_beg (item)

```

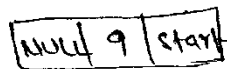
{ if AVAIL = NULL
    Write "No memory for creation of new node";
    Goto Exit;
    New_node := AVAIL;
    AVAIL := AVAIL → Next;
    New_node → prev := NULL;
    New_node → data := item;
    New_node → Next := Start;
    Start → prev := New_node;
    Start := New_node;
}

```

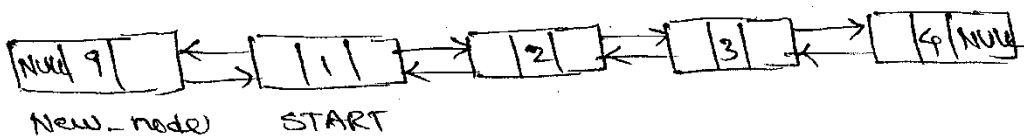
Example :- insert newnode at beginning of list with data '9'



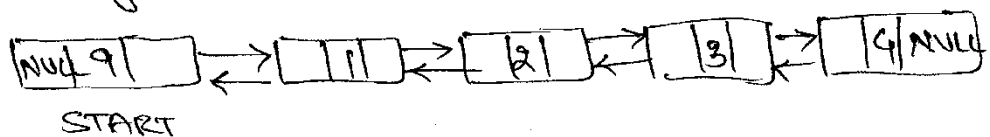
- Allocate new memory for node with data '9' and the field values



- Add the new node before the start node and change the start previous field address to new node



- Now change START position to the new node



- Finally we are inserted new_node at beginning of list.

(ii) At Ending of the list :-

Algorithm insert_end (item)

```

{ if AVAIL = NULL
    Write "No memory for creation of new node";
    Goto Exit;
    new_node := AVAIL;
    AVAIL := AVAIL → Next;

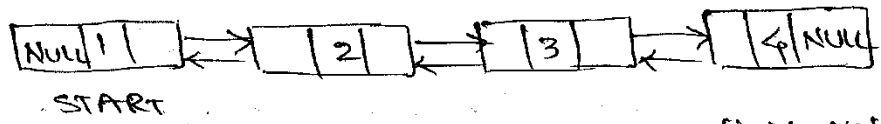
```

```

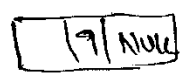
new_node → data := item;
new_node → next := NULL;
PTR := START;
While PTR → next != NULL do
    PTR := PTR → next;
PTR → next := new_node;
new_node → prev := PTR;
}

```

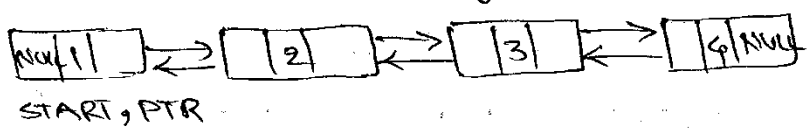
Example :- Insert new node at end of the list with data '9'



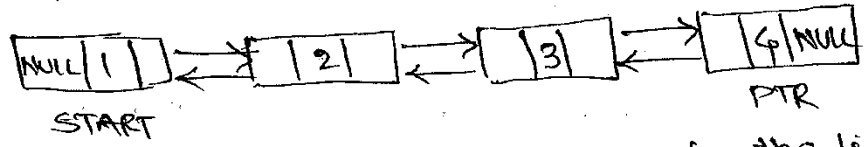
- Allocate memory for new node and field values



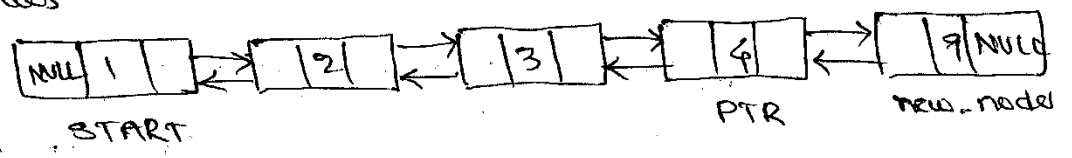
- Take PTR variable, initially it points to the first node



- Move PTR to the end of list



- Add new_node after PTR node in the list and change the PTR next field and new_node prev field values



- Finally we are inserted new node at end of the list

(ii) At particular position :-

Algorithm insert_pos (pos, item)

```

}
if AVAIL = NULL
    Write "No memory for new node";
    Goto Exit;
new_node := AVAIL;
AVAIL := AVAIL → next;
new_node → data := item;

```

PTR := START ; i = 1 ;

While i < POS - 1 do

PTR := PTR → Next ;

i++ ;

new_node → prev := PTR ;

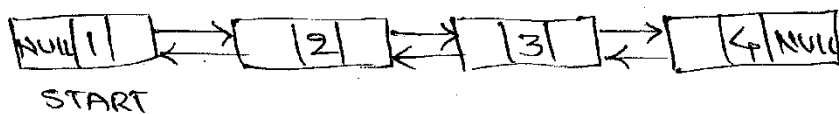
new_node → Next := PTR → Next ;

PTR → next → prev := new_node ;

PTR → next := new_node ;

}

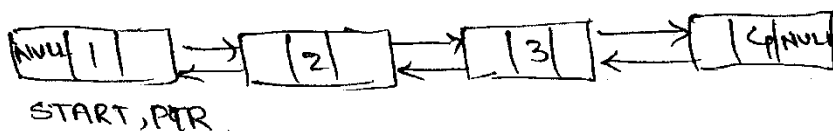
Example :- insert new node of data 9 at POS = 4.



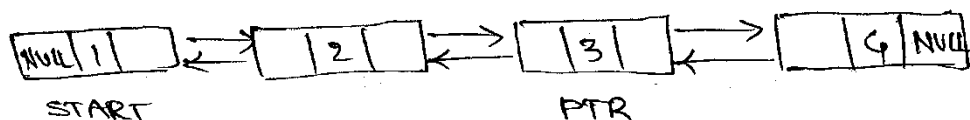
- Allocate memory for new node of data 9



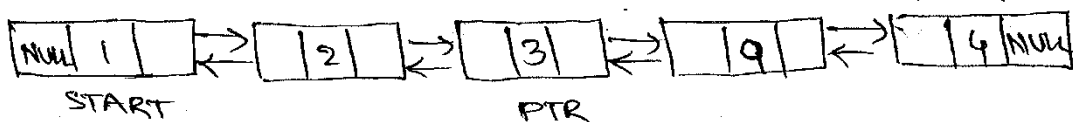
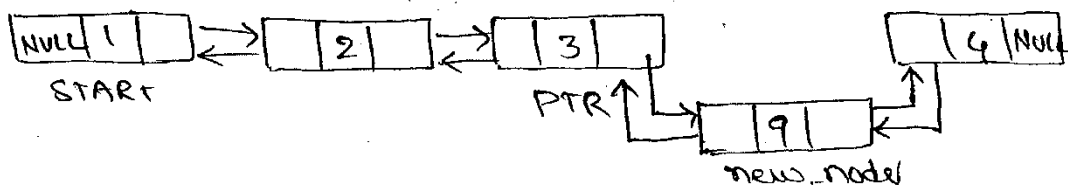
- Take PTR variable its points to the first node



- Move PTR to the next node until POS - 1 location



- Insert new node between PTR and its next node and change all the field of PTR next and new node fields



- Finally we are inserted new-node at particular position of 4.

2. Deletion :-

- We can delete the node from double linked list in 3 ways

(i) At beginning of the list

(ii) At End of the list

(iii) At particular position of the list.

(i) Delete Node at beginning of the list :-

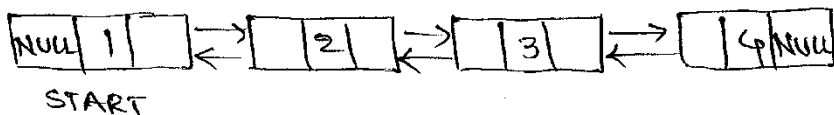
Algorithm delete_beg ()

```

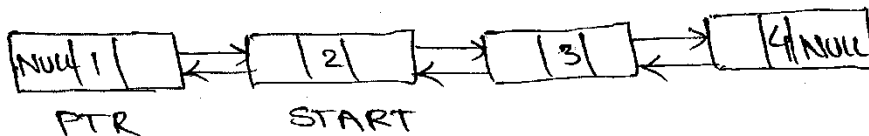
{
  if START = NULL
    write "No nodes in the list";
    Go to Exit;
  PTR := START;
  START := START -> Next;
  START -> Prev := NULL;
  free ( PTR );
}

```

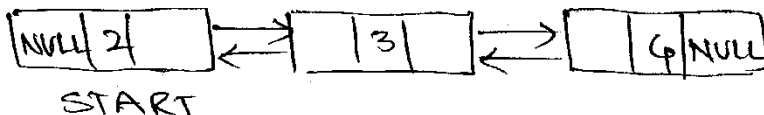
Example :- Delete the first node from double linked list.



- Takes PTR its points to the first nodes and change START value to the next node



- Delete PTR node from the list and change START previous field value to NULL.



- Finally we are deleted the beginning of the node from the given double linked list.

(ii) Delete Node at End of the list :-

Algorithm delete_End ()

```

{
  if START = NULL
    write "No nodes in the list";
    Go to Exit;
  PTR := START;
  while PTR -> Next != NULL do
    PTR := PTR -> Next;
}

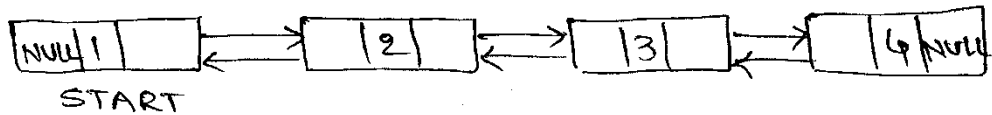
```

PTR → prev → Next := NULL

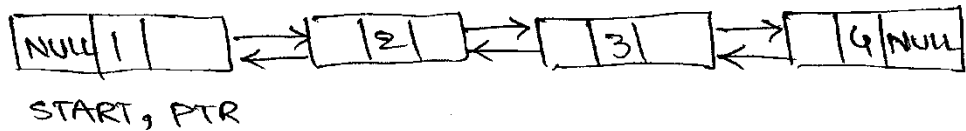
free (PTR);

}

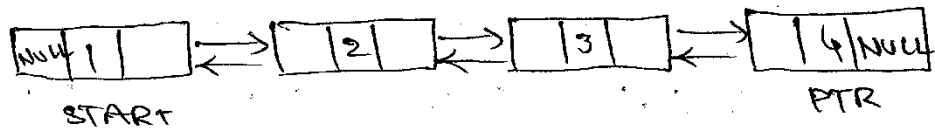
Example :- Delete node at last from the given list



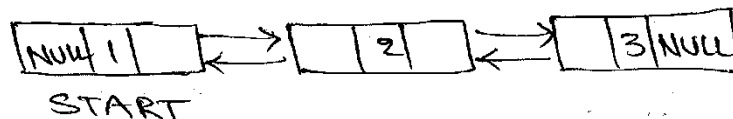
- Take PTR variable its pointing to the START node



- Move PTR to end of the list.



- Change PTR previous node next field and delete PTR node from list



- Finally we are deleted the last node from list.

(ii) Delete node at given position :-

Algorithm Delete_Pos (POS)

{

if START = NULL

Write "No nodes in the list";

Goto Exit;

PTR := START;

i := 1;

While i < POS do

PTR := PTR → Next;

PTR → prev → Next := PTR → Next;

PTR → Next → prev := PTR → prev;

free (PTR);

}

Example :- Delete a node at given position of '3' from the list

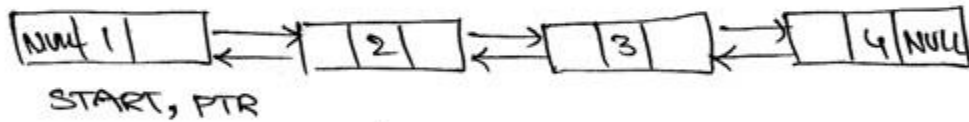
(20)

Example :- Delete a node at given position of '3' from the list

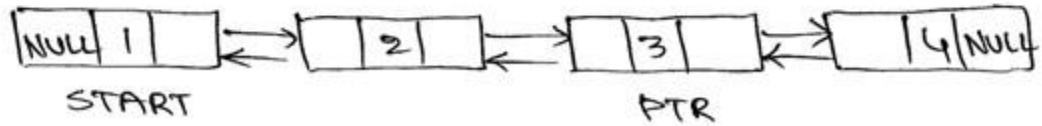
(20)



- Take PTR and it is pointing to START node

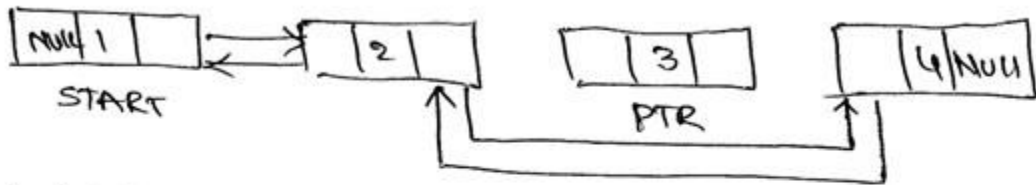


- move PTR to upto given position of '3'.

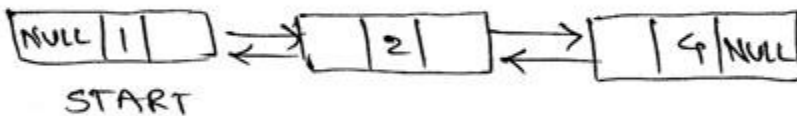


1d

- Now change the PTR node previous node next field and PTR next node previous field values



- Now delete PTR node from the list



- Finally we are deleted the given position node from the list.

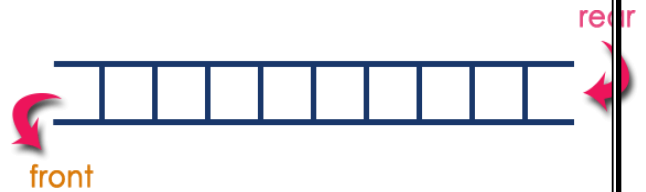
Unit – III

Syllabus:

- **Queues:** Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.
- **Stacks:** Introduction to Stacks, Array Representation of Stacks, Operations on Stacks, Linked list Representation of Stacks, Operations on Linked Stack, Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

QUEUE:

- Queue is a linear data structure in which elements can be inserted from one end called **rear** and deleted from other end called **front**.
- The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue respectively. The first element that gets added into the queue is the first one to get removed from the queue. Hence the queue is referred to as First-In-First-Out list (FIFO).



Operations performed on Queue:

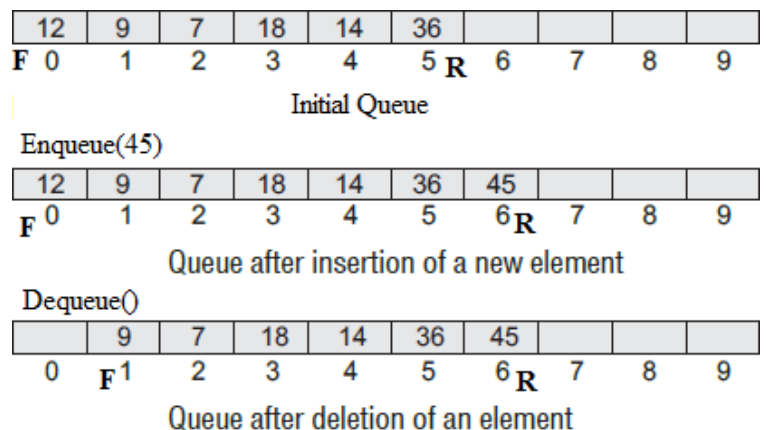
There are two possible operations performed on a queue. They are

- ✓ enqueue: Allows inserting an element at the rear of the queue.
- ✓ dequeue: Allows removing an element from the front of the queue.

REPRESENTATION OF QUEUES:

ARRAYs: Queues can be easily represented using linear arrays. Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown

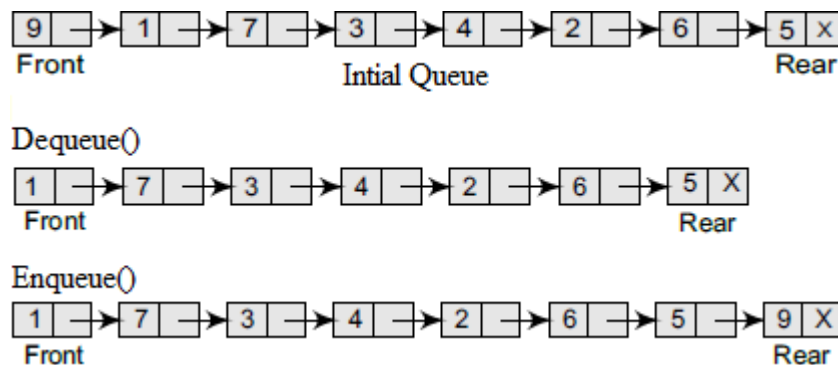
Drawback: The array must be declared to have some fixed size. If we allocate space



for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted.

LINKED LISTS:

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.
- The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end.
- If FRONT = REAR = NULL, then it indicates that the queue is empty.



IMPLEMENTATION OF QUEUES:

Using Arrays:

Algorithm for ENQUEUE operation

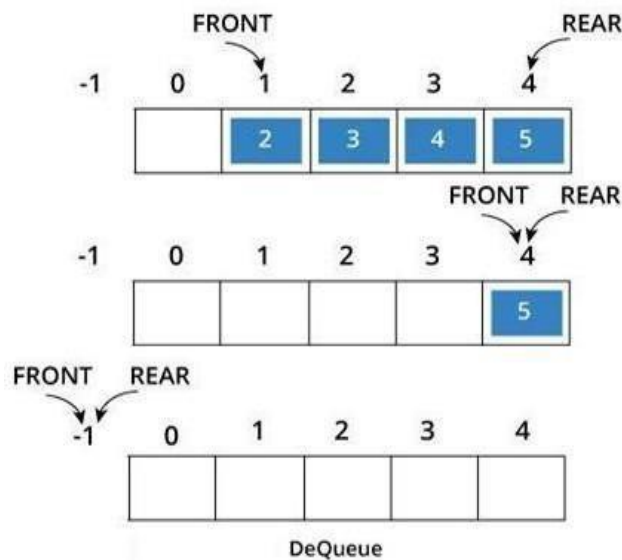
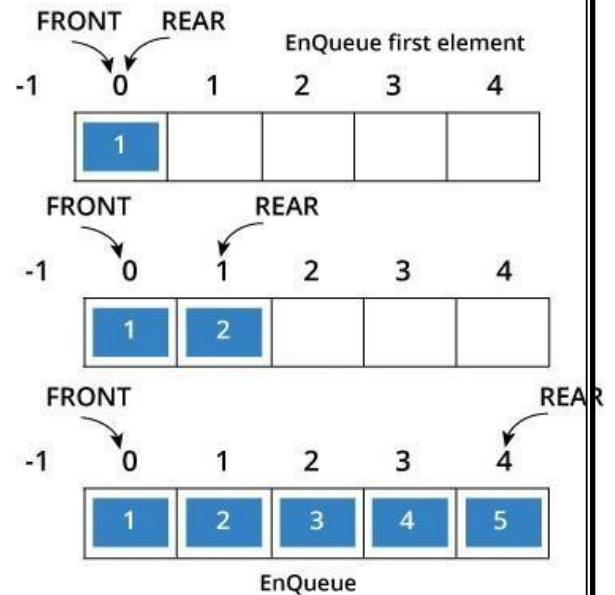
1. Check whether queue is FULL. (**rear** >= **SIZE-1**)
2. If it is **FULL**, then display an error message "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
3. If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

Algorithm for DEQUEUE operation

1. Check whether queue is EMPTY. (**front** == **-1**)
2. If it is **EMPTY**, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
3. If it is **NOT EMPTY**, then display **queue[front]** as deleted element, increment the front value by one (**front ++**). If we are deleting last element both front and rear are equal (**front == rear**), then set both front and rear to '-1' (**front = rear = -1**).

Implementation:

- Let us consider a queue, which can hold maximum of five elements.
- Initially the queue is empty. An element can be added to the queue only at the rear end of the queue.
- Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the



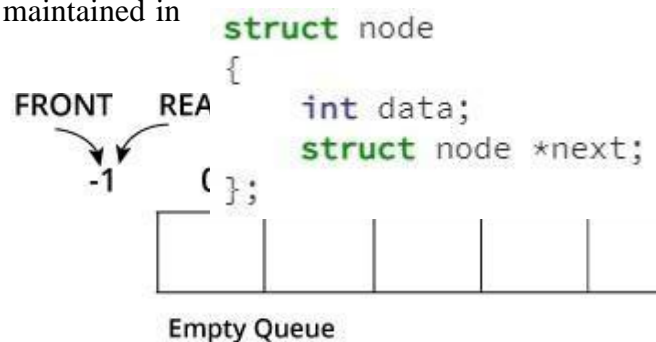
rear end. If

we are inserting first element into the queue then change front to 0 (Zero).

- Now, delete an element 1. The element deleted is the element at the front of the queue. So the status of the queue is:
 - When the last element delete 5. The element deleted at the front of the queue. So the status of the queue is empty. So change the values of front and rear to -1 (**front=rear= -1**)
- The dequeue operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.
- Drawback:** If we implement the queue using an array, we need to specify the array size at the beginning (at compile time). We can't change the size of an array at runtime. So, the queue will only work for a fixed number of elements.

Using Linked List:

- In a linked queue, each node of the queue consists of two parts i.e. data part and the next part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.



- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty. Initially

```
struct node *front = NULL, *rear = NULL;
```

Operation on Linked Queue: There are two basic operations which can be implemented on the linked queues. The operations are Enqueue and Dequeue.

Enqueue function: Enqueue function will add the element at the end of the linked list.

1. Declare a new node and allocate memory for it.
2. If front == NULL, make both front and rear points to the new node.
3. Otherwise, add the new node in rear->next (end of the list) and make the new node as the rear node. i.e. rear = new node

Dequeue function: Dequeue function will remove the first element from the queue.

1. Check whether the queue is empty or not
 2. If it is the empty queue (front == NULL), We can't dequeue the element.
 3. Otherwise, Make the front node points to the next node. i.e front = front->next;
- if front pointer becomes NULL, set the rear pointer also NULL.

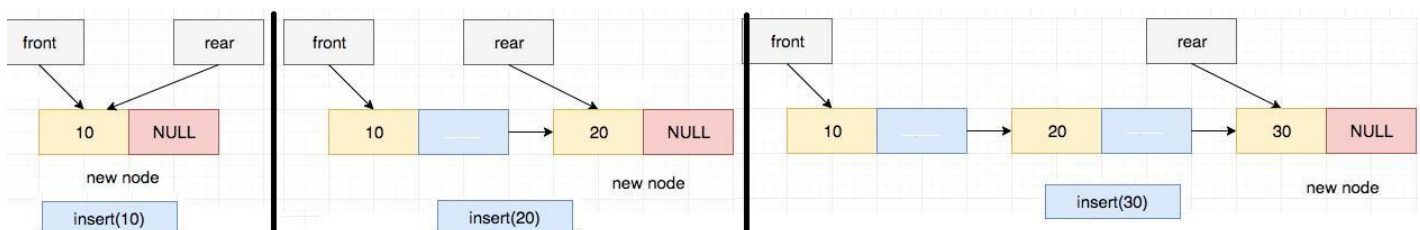
Free the front node's memory.

```
void dequeue()
{
    struct node *ptr;
    if(front == NULL)
        printf("Queue is Empty");
    else
    {
        ptr = front;
        front = front->next;
        free(ptr);
        if(front == NULL)
            rear = NULL;
    }
}
```

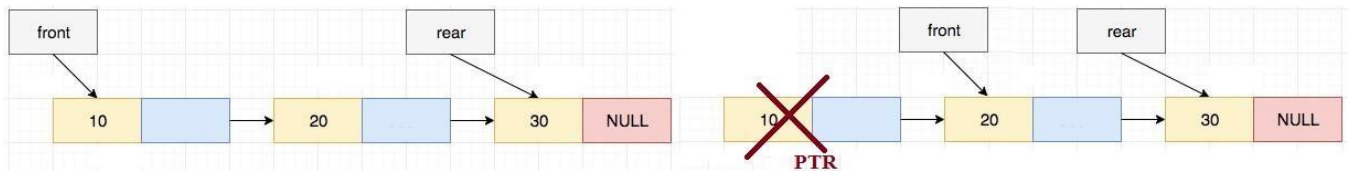
```
void enqueue(int value)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;

    if(front == NULL && rear == NULL)
        front = rear = newNode;
    else
    {
        rear->next = newNode;
        rear = newNode;
    }
}
```

Example: Enqueue()



Dequeue()



TYPES OF QUEUES:

A queue data structure can be classified into the following types:

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queue

CIRCULAR QUEUES:

- In a Linear queue, once the queue is completely full, it's not possible to insert any more elements. When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, but **rear** is still pointing to the last element of the queue, we cannot insert new elements.
- **Circular Queue** is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Operations on Circular Queue: The following are the operations that can be performed

- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the **rear end**.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the **front end**.

Enqueue operation: The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- From 2nd element onwards, When we insert a new element, the rear gets incremented, i.e., $rear=rear+1$.

Queue is not full:

- **If $rear \neq \max - 1$,** then rear will be incremented and the new value will be inserted at the rear end of the queue.
- **If $front \neq 0$ and $rear = \max - 1$,** it means that queue is not full, then set the value of rear to 0 and insert the new element there.

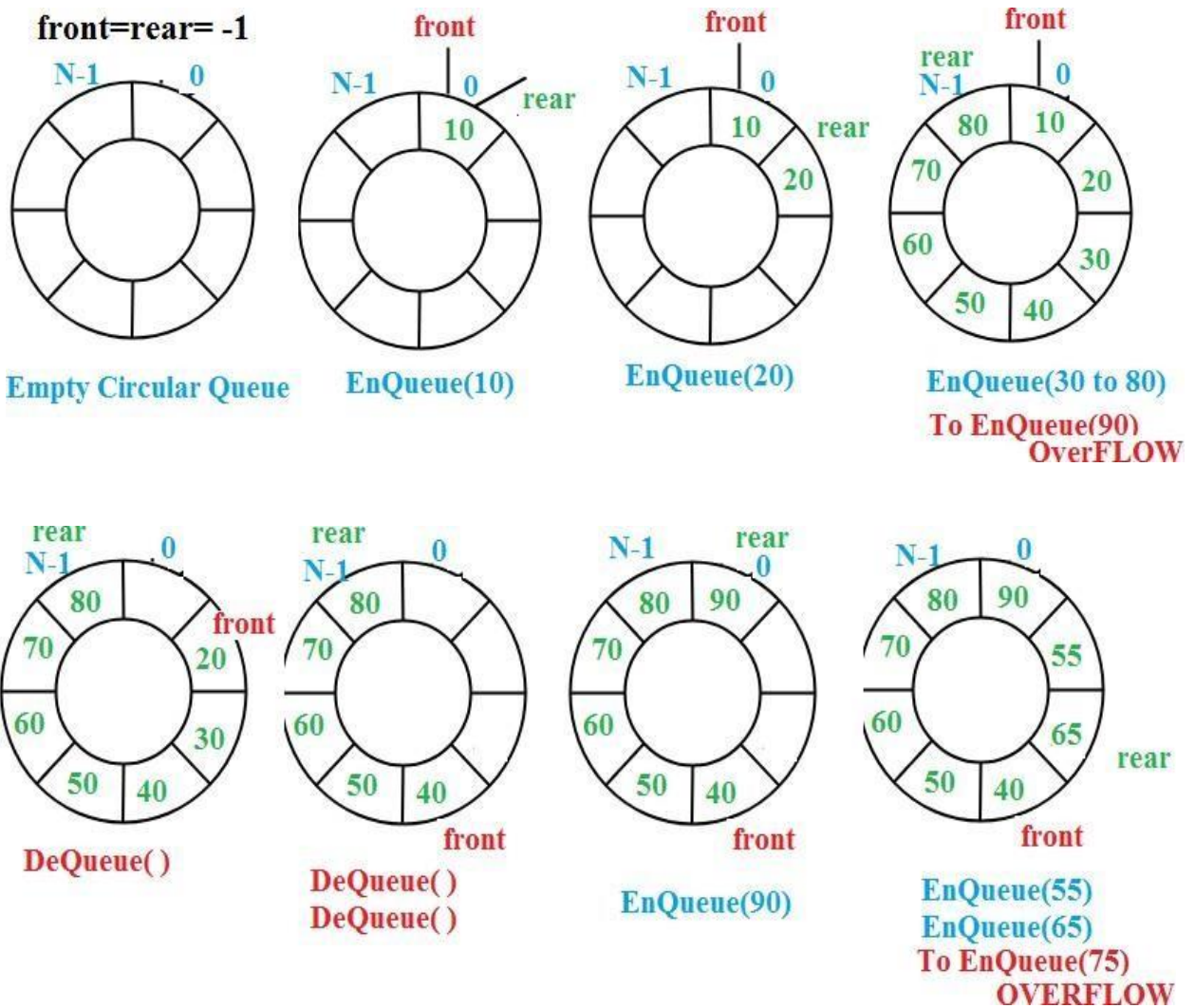
Queue is full:

- When **$front == 0$ && $rear = \max - 1$,** which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- $front == rear + 1$;

Dequeue Operation: The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset -1.

Let's understand the enqueue and dequeue operation through the diagrammatic representation.

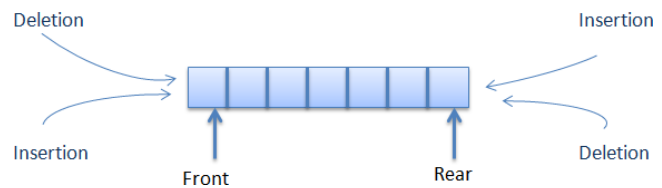


Applications of Queue:

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used to transfer data asynchronously between two processes
3. Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
4. Queues are used in Playlist for jukebox to add songs to the end, play from the front.
5. Queues are used in operating system for handling interrupts. The interrupts are handled in the same order as they arrive i.e First come first served.

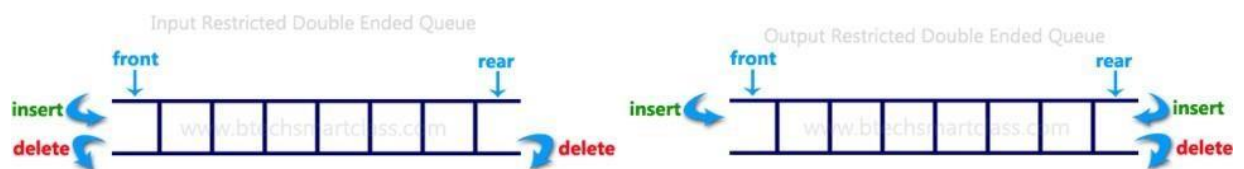
DEQUE:

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).



Types of Deque:

1. **Input Restricted Deque:** In this deque, input is restricted at a single end but allows deletion at both the ends.
2. **Output Restricted Deque:** In this deque, output is restricted at a single end but allows insertion at both the ends.



Operations on a Deque

- Initially take an array (deque) of size **n**. and Set two pointers at the first position and set **front = -1** and **rear = -1**.
1. **Insert at the Front:** This operation adds an element at the front.
 - Check the position of front, If $\text{front} < 1$, we can't add elements in the front end. Otherwise decrement the front and at front location we can insert the element.
 2. **Insert at the Rear:** This operation adds an element to the rear.
 - Check if the array is full. Then the queue is overflow. Otherwise, reinitialize $\text{rear} = 0$ & $\text{front} = 0$ for the first insertion, Else, increase rear by 1. and at rear location we can insert the element.
 3. **Delete from the Front:** The operation deletes an element from the front.
 - Check If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition). If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$. Else, $\text{front} = \text{front} + 1$.
 4. **Delete from the Rear:** This operation deletes an element from the rear.
 - If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition). If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$. Else, $\text{rear} = \text{rear} - 1$.

Priority Queue:-

- A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.
- The general rules of processing the elements of a priority queue are
 - An element with higher priority is processed before an element with a lower priority.
 - Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

Array Representation of a Priority Queue:

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.
- FRONT[P] and REAR[P] contain the front and rear values of row P, where P is the priority number.

		SIZE of CQ					FRONT	REAR
		0	1	2	3	4		
p r i o r i t y	1	A					2	2
	2					F	4	4
	3	B		C	D		1	3
	4	E					0	0
	5	G		H	I	J	1	4
	6	M			K	L	3	0

Insertion:

- To insert a new element with priority P in the priority queue, add the element at the rear end of row P, where P is the row number as well as the priority number of that element.
- For example, if we have to insert an element X with priority number 2, then the priority queue will be given as shown in Fig.

ENQUEUE		SIZE of CQ					FRONT	REAR
		0	1	2	3	4		
p r i o r i t y	1	A				Y	2	3
	2	X				F	4	0
	3	B		C	D		1	3
	4	E					0	0
	5	Z	G	H	I	J	1	0
	6	M			K	L	3	0

DEQUEUE		SIZE of CQ					FRONT	REAR	
		0	1	2	3	4			
p r i o r i t y	1	Y					3	3	A
	2	X					0	0	F
	3	C		D			2	3	B
	4	E					0	0	
	5	Z	G	H	I	J	1	0	
	6	M			L		4	0	K

Deletion:

- To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue.
- In our priority queue, the first non-empty queue is the one with priority number 6 and the front element is K, so K will be deleted and processed first.

Multiple Queues:-

- When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.
- An array `Queue[n]` is used to represent two queues, Queue A and Queue B. The value of `n` is such that the combined size of both the queues will never exceed `n`. While operating on these queues, it is important to note one thing—queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

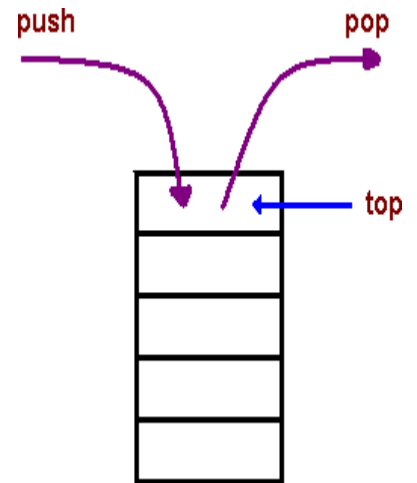
Example:

	QA	10	20	30	40	35	25	15	5	QB	
EMPTY	fA=rA = -1	fA			rA		rB			fB	fB=rB= SIZE 10
First Insertion	fA=rA = 0										fB=rB= SIZE-1

- In the above example the array consists two queues like QA and QB. For QA there are pointers like fA(front of QA) and rA(rear of A). similarly for QB are fB & rB.
- Initially for QA, the pointer values of fA=rA= -1. For QB, the pointer values are fB=rB=SIZE. Because initially QA and QB are empty.
- For the first insertion in QA, the values of fA=rA=0. Similarly for QB, the values are fB=rB=SIZE-1.
- From the second insertion onwards we can increment only the rear pointer rA for QA and decrement the rear rB for QB.
- Delete the elements from queue only at front end. In QA, the elements can delete from fA, if you delete the element then increment fA. In QB, the elements can delete from fB, if you delete the element then decrement fB.
- When the condition $rA=rB-1$ or $rA+1=rB$ meets then the entire queue is full. If you try to insert the element in either of queues it says that QUEUE is OVERFLOW.

Stack:-

- Stack is a linear data structure in which insertion and deletion can perform at the same end called **top** of stack.
- When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop.
- Stack is also called as Last-In-First-Out (LIFO) list which means that the last element that is inserted will be the first element to be removed from the stack.
- When a stack is completely full, it is said to be Stack is **Overflow** and if stack is completely empty, it is said to be Stack is **Underflow**.



REPRESENTATION & IMPLEMENTATION STACK:

Array Representation of Stacks:

- Every stack has a variable called TOP associated with it, which is used to pointing the topmost element of the stack. It is this position where the element will be inserted to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If $TOP = NULL$, then it indicates that the stack is empty and if $TOP = MAX-1$, then the stack is full.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	TOP = 4	5	6	7	8	9

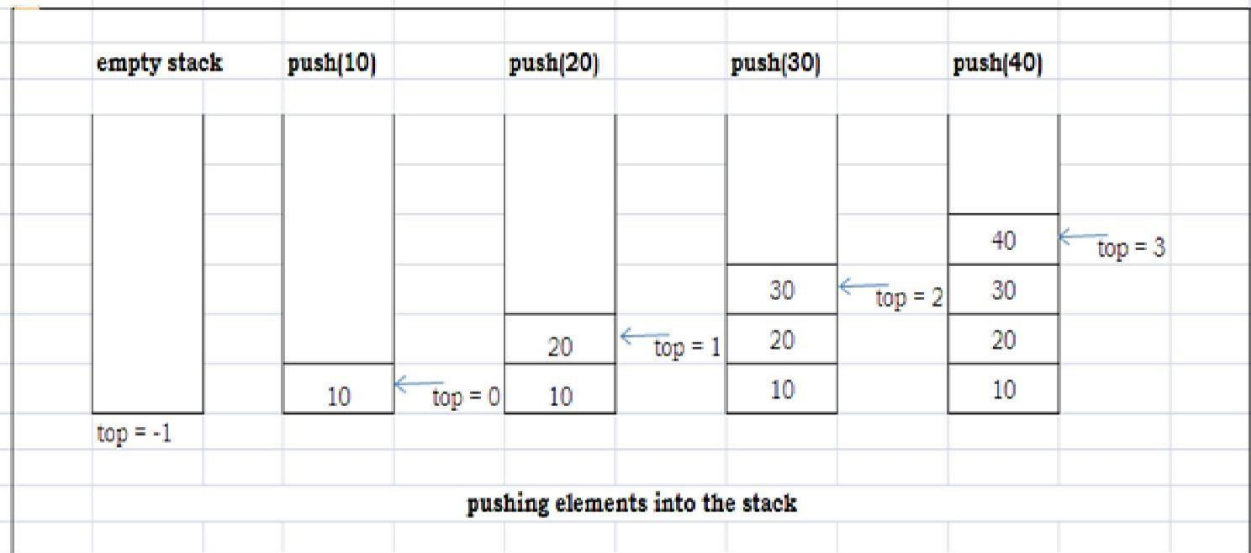
Array Implementation of Stack:

The basic operations performed in a Stack:

1. Push(x) - add element x at the top of the stack
2. Pop() - remove top element from the stack
3. peek() - get top element of the stack without removing it

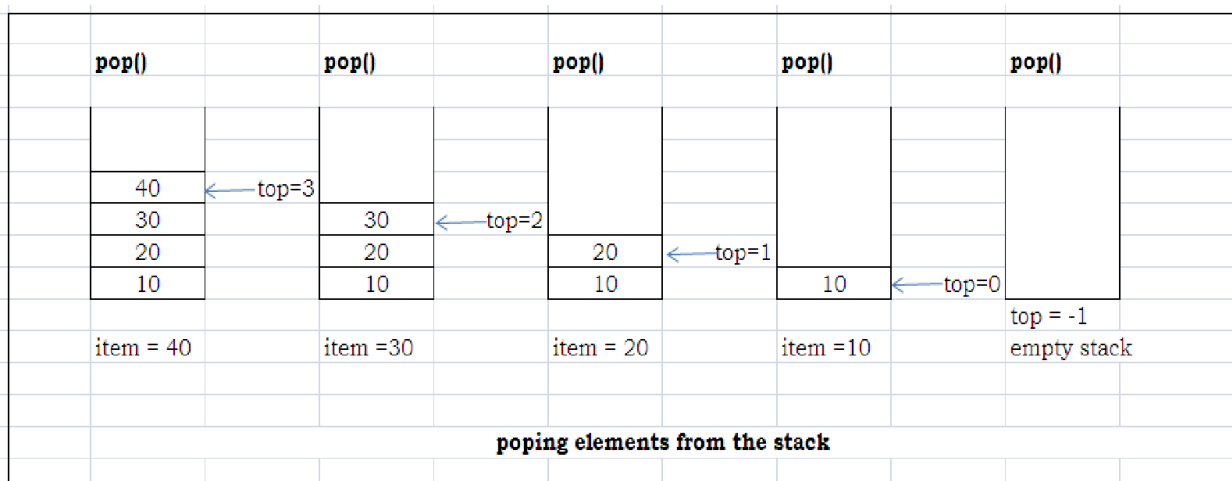
Algorithm for PUSH operation:

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element at top location.



Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.



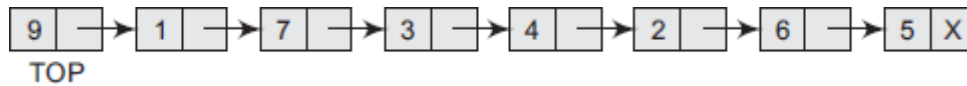
Algorithm for PEEK operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top without removing it.

Linked Representation of Stacks:

- The drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance
- In a linked stack, every node has two parts, one that stores *data* and another that stores the *address* of the next node. The START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the TOP (similar to **insertion at beginning**).

- If TOP = NULL, then it indicates that the stack is empty.
- The linked representation of a stack is



Linked Implementation of Stack:

- In a linked stack, each node of the stack consists of two parts i.e. data part and the next part. Each element of the stack points to its immediate next element in the memory.
- In the linked stack, there one pointer maintained in the memory i.e. TOP pointer. The TOP pointer contains the address of the starting element of the STACK.
- Both Insertion and deletions are performed at only one end called TOP. If TOP is NULL, it indicates that the stack is empty. Initially

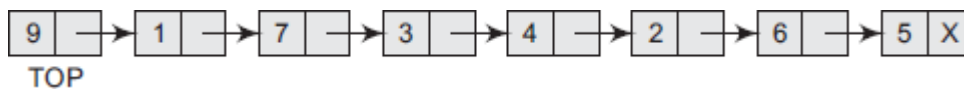
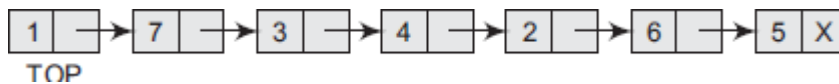
```
struct node
{
    int data;
    struct node *next;
};
```

```
struct node *TOP = NULL ;
```

Operation on Linked STACK: There are two basic operations which can be implemented on the linked queues. The operations are PUSH and POP.

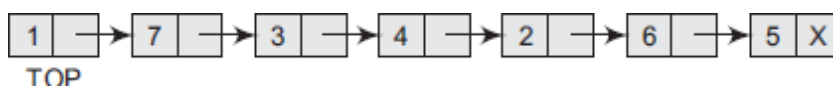
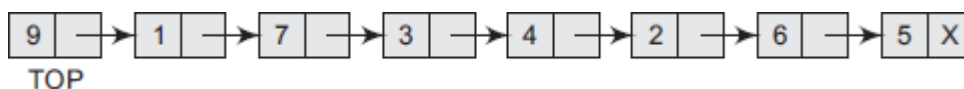
PUSH function: PUSH function will add the element at the beginning of the linked list.

1. Declare a new node and allocate memory for it.
2. If TOP == NULL, make TOP points to the new node.
3. Otherwise, add the new node at TOP end and makes the next of new node is previous TOP.



POP function: POP function will remove the TOP element from the STACK.

1. Check whether the stack is empty or not
2. If it is the stack is empty (TOP == NULL), We can't POP the element.
3. Otherwise, Make the TOP node points to the next node. i.e TOP = TOP->next; Free the TOP node's memory.



Algorithms:

void PUSH(int Value)

```
{
struct node *new_node=(struct node *)
malloc(sizeof(struct node));

new_node->data = Value;
new_node->Next = TOP
TOP = new_node;
}
```

void POP()

```
{
if(TOP==NULL)
printf("UNDERFLOW")
else
{
PTR=TOP;
TOP=TOP->Next;
free(PTR)
}
}
```

Applications of Stacks

- ✓ Stack is used to reversing the given string.
- ✓ Stack is used to evaluate a postfix expression.
- ✓ Stack is used to convert an infix expression into postfix/prefix form.
- ✓ Stack is used to matching the parentheses in an expression.

Reversing list:

A list of numbers or string can be reversed by using the stack, perform the following steps

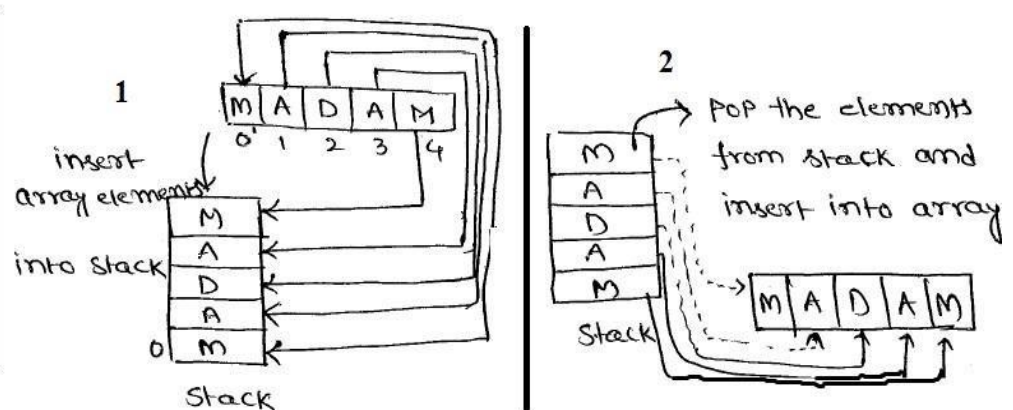
1. Reading the elements from the array and pushed into the stack.
2. Pop the elements and again stored into the array starting from first index

Algorithm:

```
void reverse()
{
// input: array A[ ], size N
// top = -1, Stack S
// output: Reversing array A[ ]
for i=0 to N-1
TOP = TOP+1;
S[TOP] = A[i];

for i=0 to N-1
A[i]=S[TOP];
TOP = TOP-1;
}
```

Example:



Factorial Calculation:

- To find the solution of larger problem, a general method is reduce the larger problem into one or more sub problems. This process will continuous until the sub problem is finding the solution. Finally using all the sub problems solutions we will find the solution for the larger problem.

- A *Recursion* is defined as a function that calls itself.
- To understand recursion, let us take an example of calculating factorial of a number.
- To calculate $n!$, we multiply the number with factorial of the number that is 1 less than that number. $n! = n*(n-1)*(n-2)* \dots *2*1$

- In other words, $n! = n * (n-1)!$
- Let us say we need to find the value of $5!$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

- This can be written as $5! = 5 * 4!$,

$$\text{where } 4! = 4 * 3!$$

Therefore, $5! = 5 * 4 * 3!$

Similarly, $5! = 5 * 4 * 3 * 2!$

Expanding further $5! = 5 * 4 * 3 * 2 * 1!$

We know, $1! = 1$

- Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function,

Base case is when $n = 1$, because if $n = 1$, the result will be 1 as $1! = 1$.

Recursive case of the factorial function will call itself but with a smaller value of n , this case can be given as $\text{factorial}(n) = n \times \text{factorial}(n-1)$

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Evaluation of Expressions:-

- An expression is defined as the combination of operands (variables, constants) and operators arranged as per the syntax of the language.
- An expression can be represented using three different notations. They are infix, postfix and prefix notations:

Prefix: An arithmetic expression in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation. Example: + A B

Infix: An arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. Example: A + B

Postfix: An arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation OR reverse polish notation.

Example: A B +

Operator Precedence: When an expression consist different level of operators we follow it. We consider five binary operations: +, -, *, / and ^ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest): ^, *, /, +, -

Operator Associativity: When an expression consist more than one same level precedence operators we follow it.

Basically we have Left to Right associativity and Right to Left Associativity. Most of the operators are follows Left to Right but some of the operators are follow Right to left Associativity like Unary (+/-), ++/-- , Logical negation (!), Pointer and address (*,&), Conditional Operators and Assignment operators(=,+=-,-=,*=,/=,%=).

Example: $x = a / b - c + d * e - a * c$

Let a = 4, b = c = 2, d = e = 3 then the value of x is found as



$$= ((4 / 2) - 2) + (3 * 3) - (4 * 2) = 0 + 9 - 8 = 1$$

EVALUATION OF POSTFIX EXPRESSION:

The standard representation for writing expressions is infix notation. But the compiler uses the postfix notation for evaluating the expression rather than the infix notation. It is an easy task for evaluating the postfix expression than infix expression because there are no parentheses. To evaluate an expression we scan it from left to right. The postfix expression is evaluated easily by the use of a stack.

To evaluate a postfix expression use the following steps...

1. Read the **poststring** from left to right
2. Initialize an **empty Stack**
3. Repeat until end of the **poststring**
 - i. If the scanned character is operand, then push it on to the Stack.
 - ii. If the scanned character is operator (+, -, *, / etc.), then pop top two elements from the stack, perform the operation with the operator then push result back on to the Stack.
4. Finally! We have one element in the stack, perform a pop operation and display the popped value as **final result**.

Postfix Expression is 5 3 + 8 2 - *		
Symbol	Stack	Evaluation
Initially	 <p>Stack is empty</p>	
5	 <p>Push(5)</p>	

3	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">3</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">5</div> Push(3)	
+	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> Value1=pop() Value2=pop() Result=Value2+Value1 Push(Result)	Value1=3 Value2=5 Result=5+3=8 Push(8)
8	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> Push(8)	
2	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">2</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> Push(2)	
-	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">6</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">8</div> Value1=pop() Value2=Pop() Result=Value2-Value1 Push(Result)	Value1=2 Value2=8 Result=8-2=6 Push(6)
*	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">48</div> Value1=pop() Value2=Pop() Result=Value2*Value1 Push(Result)	Value1=6 Value2=8 Result=8*6=48 Push(48)
End of Expression	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;"> </div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 2px;">48</div> Result=pop()	Final Result is 48

Conversion of INFIX to POSTFIX:

Procedure to convert from infix expression to postfix expression is as follows.

1. Initialize an empty stack
2. Push “(“ onto Stack, and add “)” to the end of Infix string.
3. Scan the Infix string from left to right until end of the infix
 - i. If the scanned character is “(“, pushed into the stack.
 - ii. If the scanned character is “)”, pop the elements from the stack up to encountering the “(“, and add the popped elements to postfix string except parentheses.
 - iii. If the scanned character is an operand, add it to the Postfix string.
 - iv. If the scanned character is an Operator, compare the precedence of the character with the element on top of the stack. If top of Stack has lower precedence over the scanned character then push the operator into the stack else pop the element from the stack and add it to postfix string and push the scanned character to stack.

Example: a * (b + c) *d)

Token	Stack				Postfix String
	(
a	(a
*	(*			A
((*	(A
b	(*	(Ab
+	(*	(+	Ab
c	(*	(+	Abc
)	(*			abc+
*	(*			abc+*
d	(*			abc+*d
)					abc+*d*

UNIT IV Trees

Introduction Terminology

Representation of trees,

Binary trees abstract data type

Properties of binary trees

Binary tree representation

Binary tree traversals: In order, preorder, post order

Binary search trees Definition

Operations: searching BST, insert into BST, delete from a BST, Height of a BST.

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

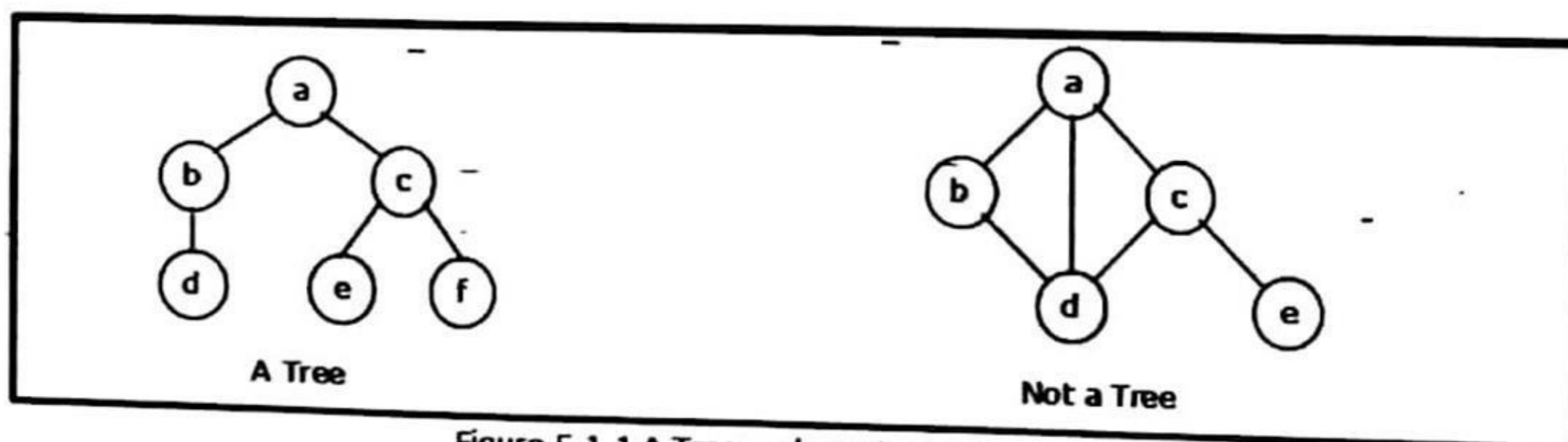


Figure 5.1.1 A Tree and a not a tree

Tree is a popular data structure used in wide range of applications. A tree data structure can be defined as follows...

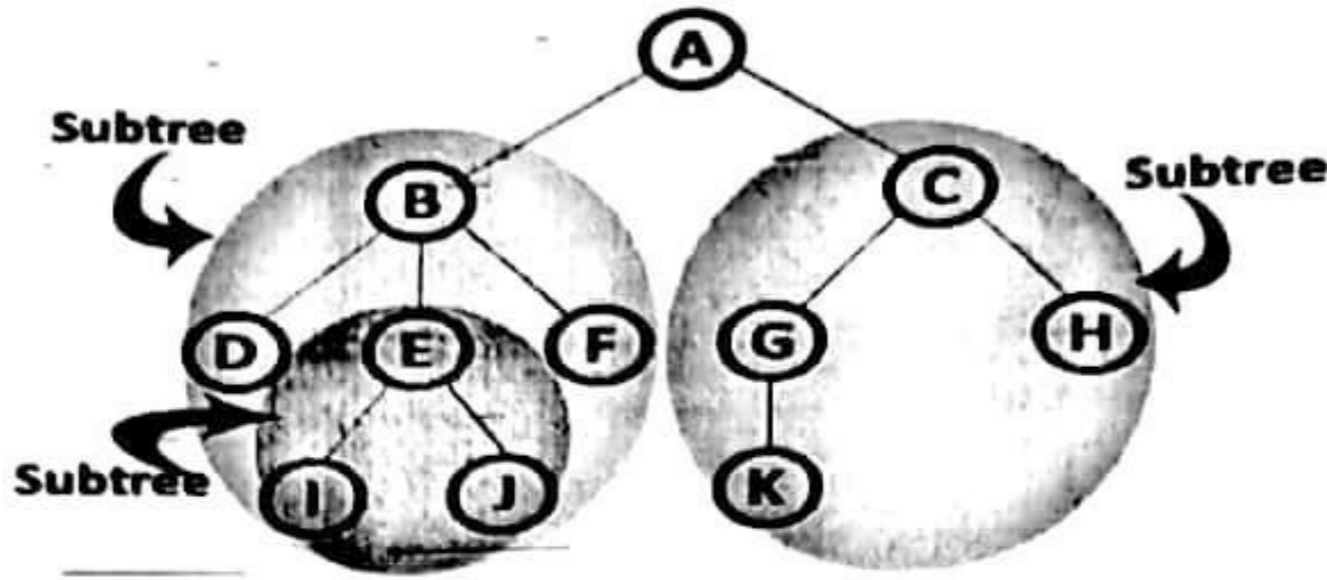
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

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

UNIT- IV

There is a specially designated node called the root. The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree. We call T_1, \dots, T_n are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

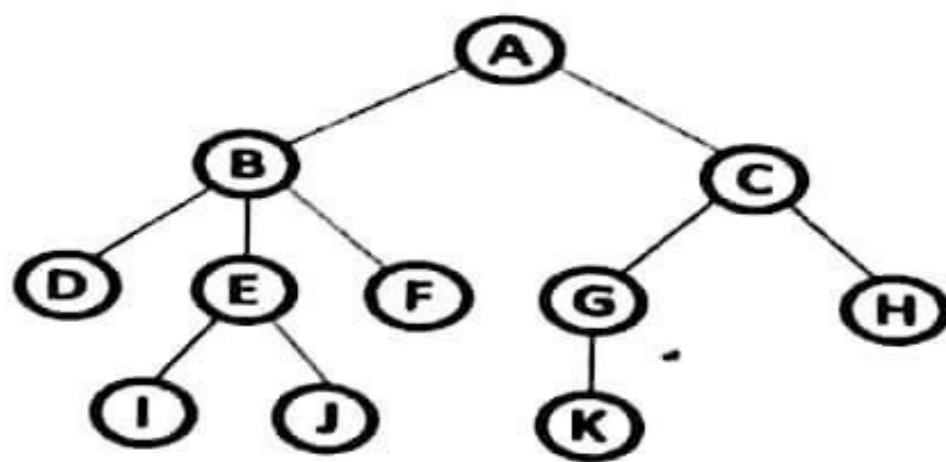
Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub trees around with minimum effort

Introduction Terminology

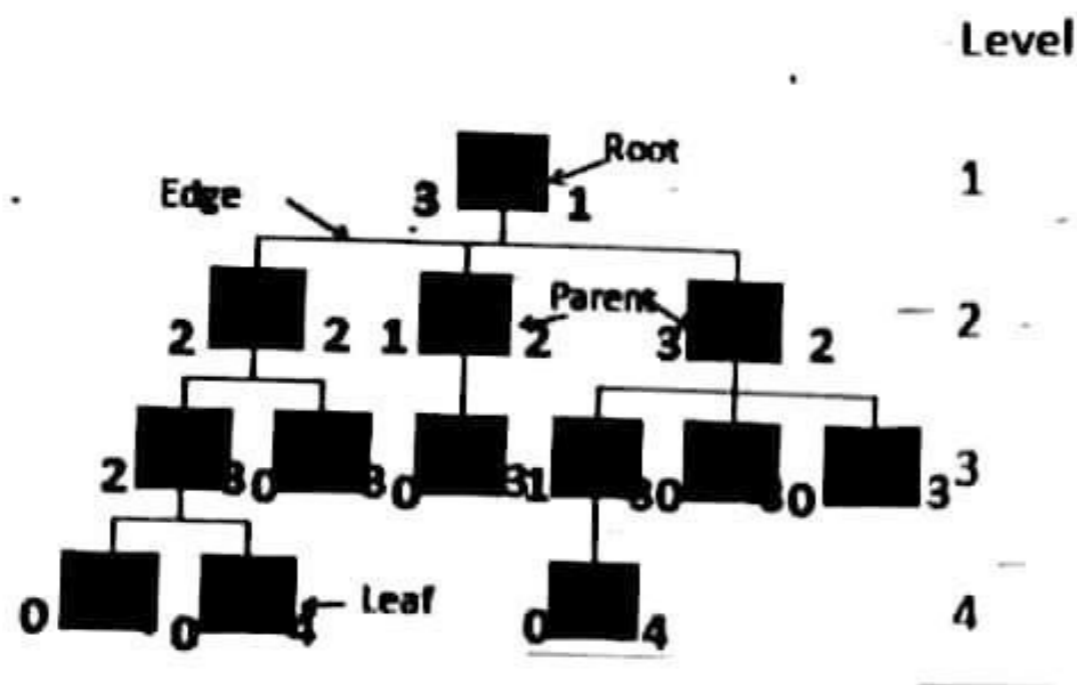
In a Tree, Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. Example



TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges

- In a tree every individual element is called as 'NODE'



1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, A is a Root node

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes. e.g., Children of D are (H, I, J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B, C, D)

6. Leaf

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node. In simple words, a leaf is a node with no child.

UNIT- IV.

In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

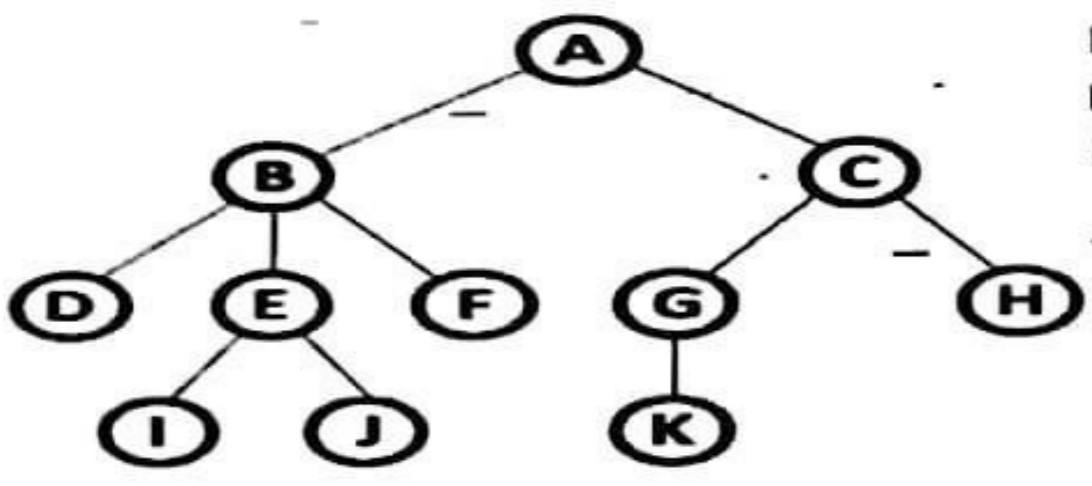
7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes. Ex: B,C,D,E,H

8. Degree

In a tree data structure, the total number of children of a node (or) number of subtrees of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 1.

10. Height

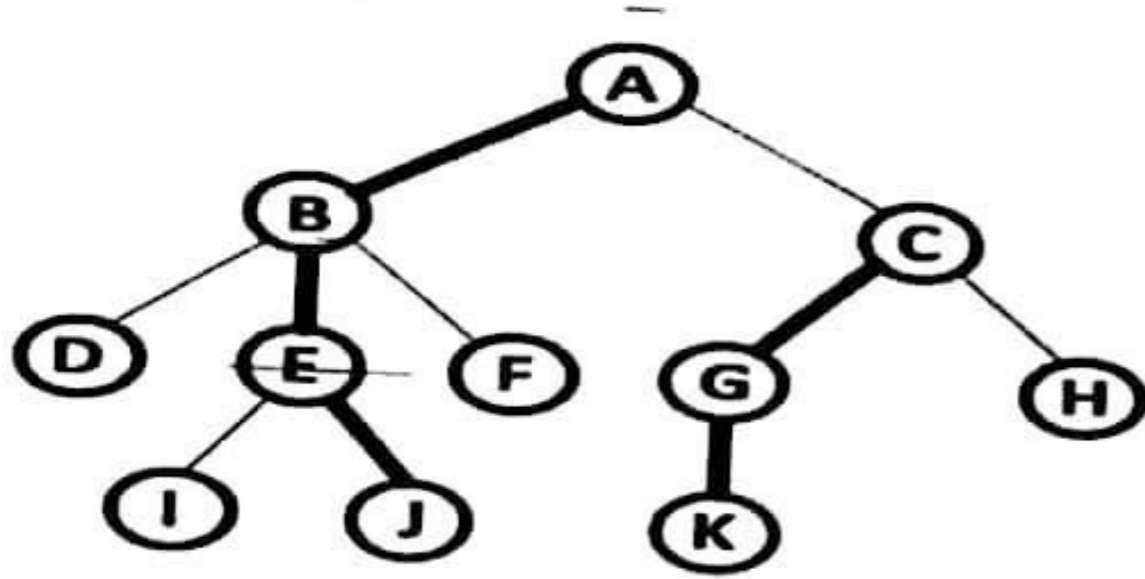
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



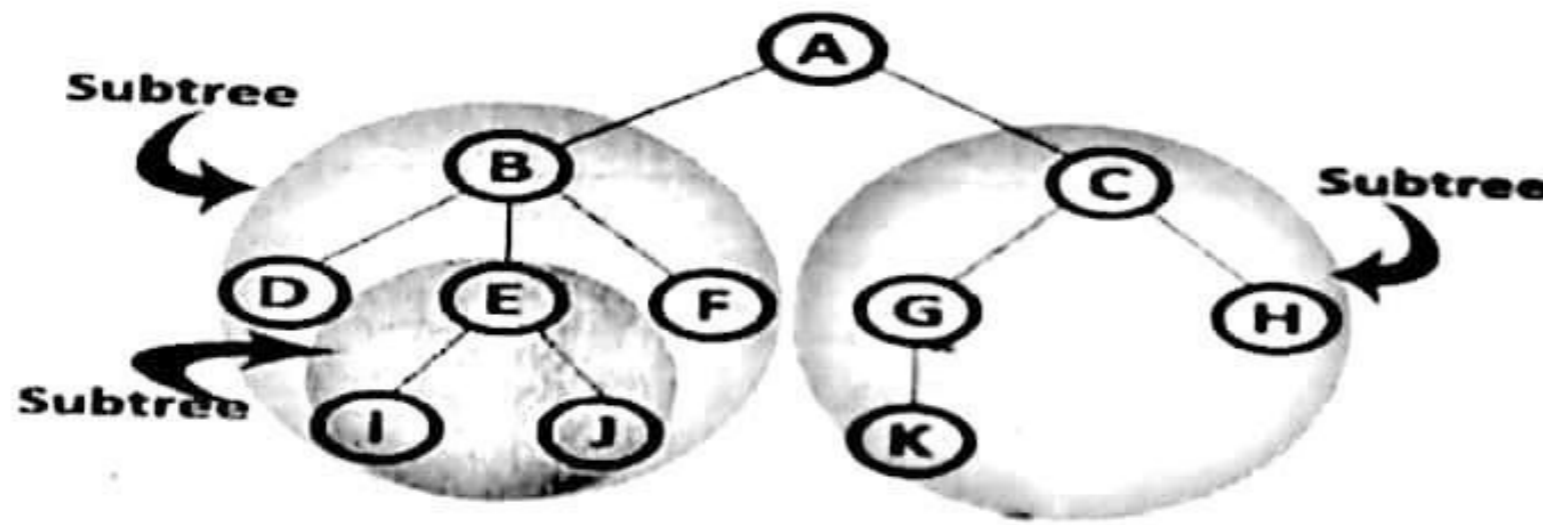
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is **A - B - E - J**

Here, 'Path' between C & K is **C - G - K**

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

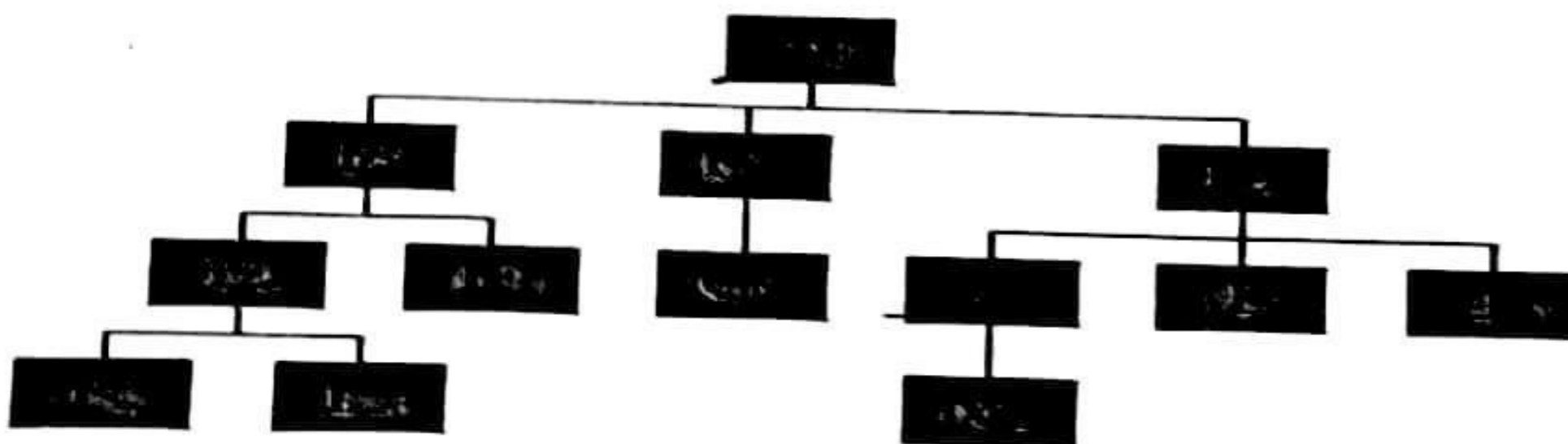


Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1. List Representation
2. Left Child - Right Sibling Representation

Consider the following tree...



UNIT- IV

1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

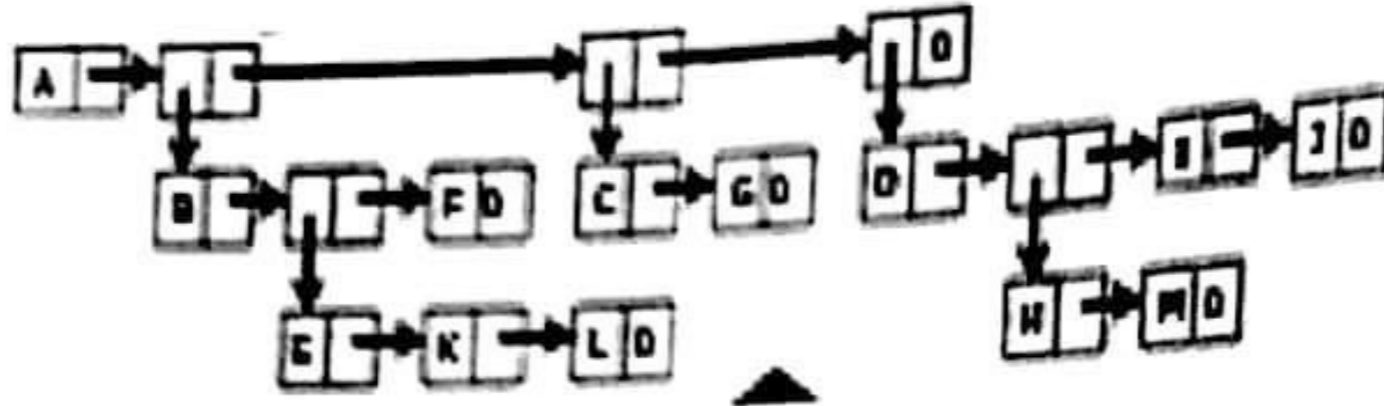


Fig: List representation of above Tree

List Representation

- (A(B(E(K,L), F), C(G), D(H(M), I, J)))
- The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link k
------	--------	--------	-----	--------

Fig: Possible node structure for a tree of degree k

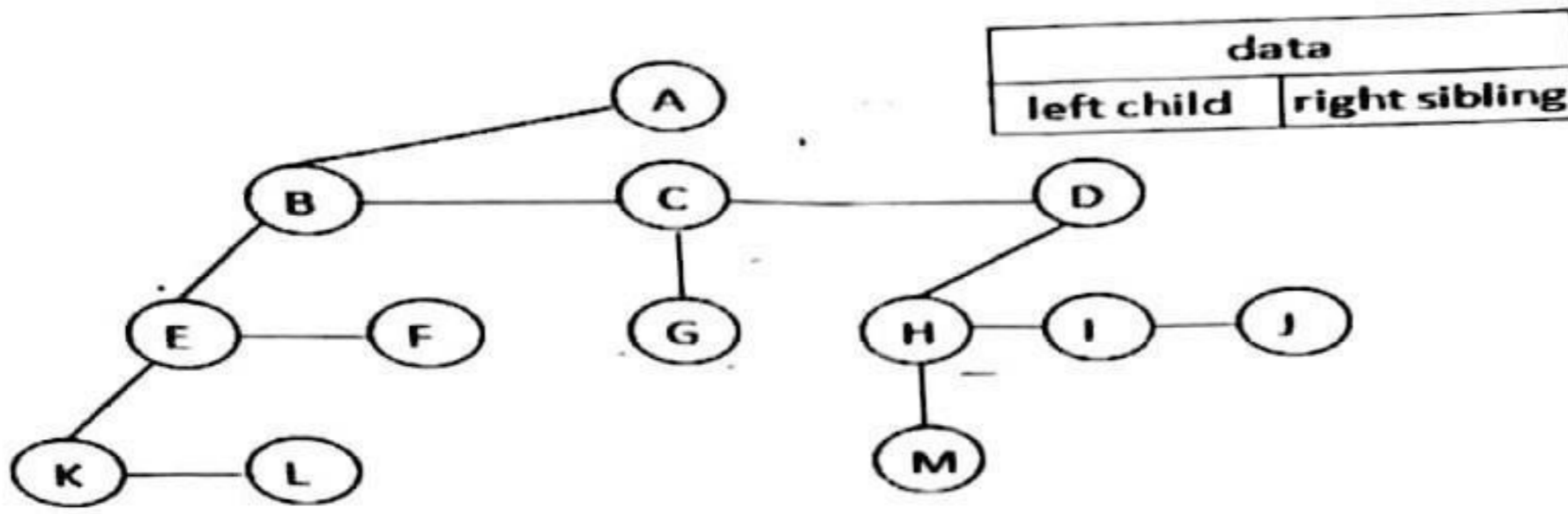
2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.

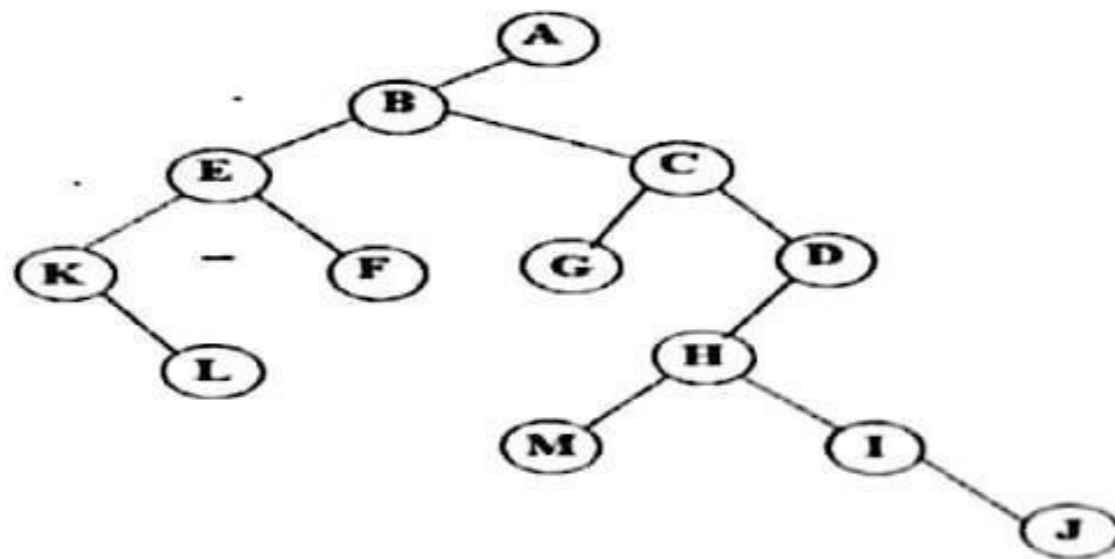
The above tree example can be represented using Left Child - Right Sibling representation as follows...



Representation as a Degree -Two-Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child- right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

Figure 5.6: Left child-right child tree representation of a tree (p.191)



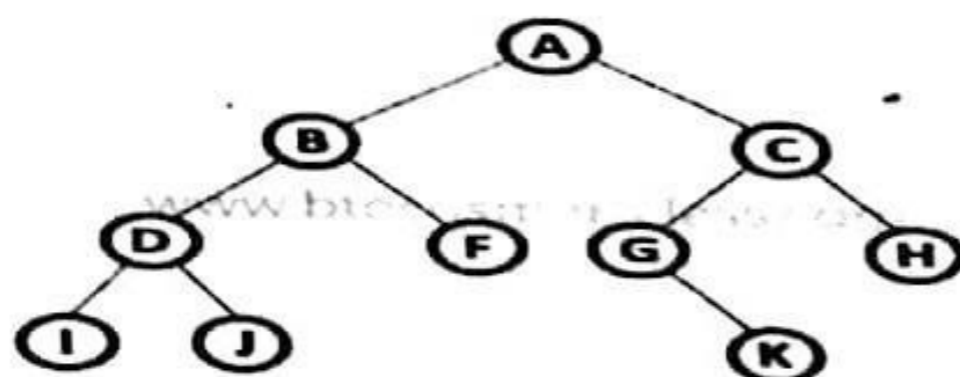
Binary Trees

Introduction

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. Example



There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

2. Complete Binary Tree

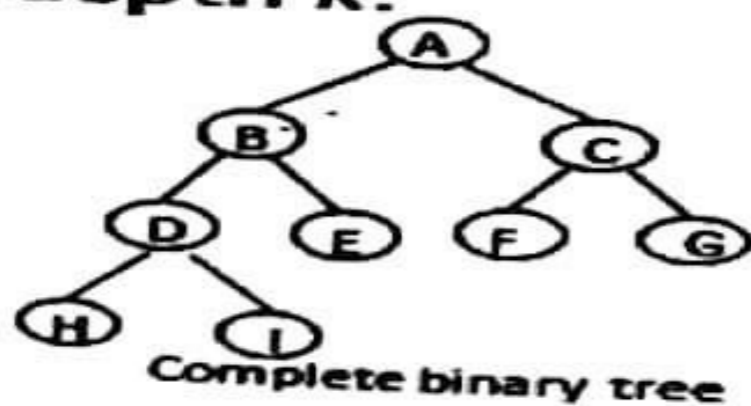
In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

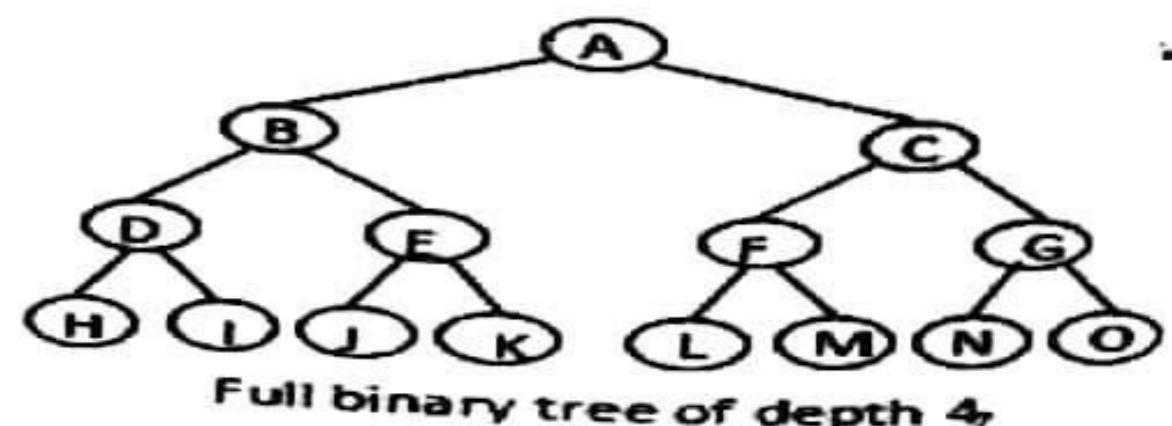
Complete binary tree is also called as Perfect Binary Tree

Full BT VS Complete BT

- A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.
- A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k .



CHAPTER 3



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

Abstract Data Type

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

ADT contains specification for the binary tree ADT.

Structure *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

Functions:

for all $bt, bt1, bt2 \in BinTree, item \in element$

BinTree Create() ::= creates an empty binary tree

Boolean IsEmpty(bt) ::= if (bt == empty binary tree) return *TRUE* else return *FALSE*

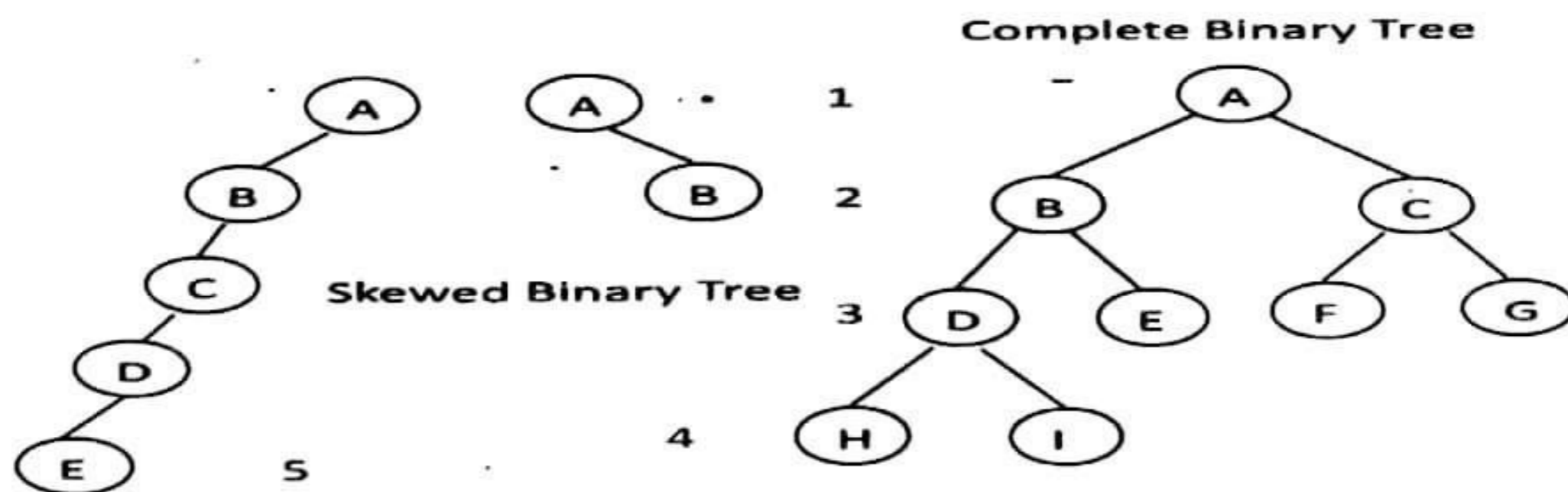
BinTree MakeBT($bt1, item, bt2$) ::= return a binary tree whose left subtree is $bt1$, whose right subtree is $bt2$, and whose root node contains the data $item$

BinTree Lchild(bt) ::= if (IsEmpty(bt)) return error else return the left subtree of bt

element Data(bt) ::= if (IsEmpty(bt)) return error else return the data in the root node of bt

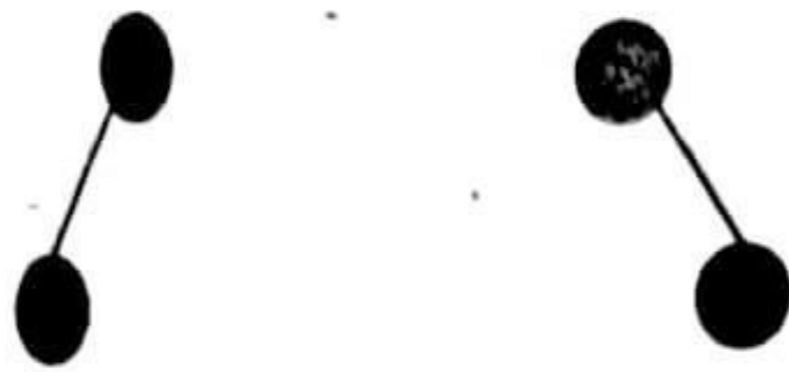
BinTree Rchild(bt) ::= if (IsEmpty(bt)) return error else return the right subtree of bt

Samples of Trees



Differences between A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



Above two trees are different when viewed as binary trees. But same when viewed as trees.

Properties of Binary Trees

1. Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof By Induction:

Induction Base: The root is the only node on level $i=1$. Hence, the maximum number of nodes on level $i=1$ is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let I be an arbitrary positive integer greater than 1. Assume that maximum number of nodes on level $i-1$ is 2^{i-2} .

Induction Step: The maximum number of nodes on level $i-1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level $i-1$, or 2^{i-1} .

The maximum number of nodes in a binary tree of depth k is $\sum_{i=1}^k 2^{i-1} = 2^k - 1$.

2. Relation between number of leaf nodes and degree-2 nodes: For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

PROOF: Let n and B denote the total number of nodes and branches in T . Let n_0, n_1, n_2 represent the nodes with zero children, single child, and two children respectively.

$$B+1=n \rightarrow B=n_1+2n_2 \implies n_1+2n_2+1=n,$$

$$n_1+2n_2+1=n_0+n_1+n_2 \implies n_0=n_2+1$$

3. A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k .

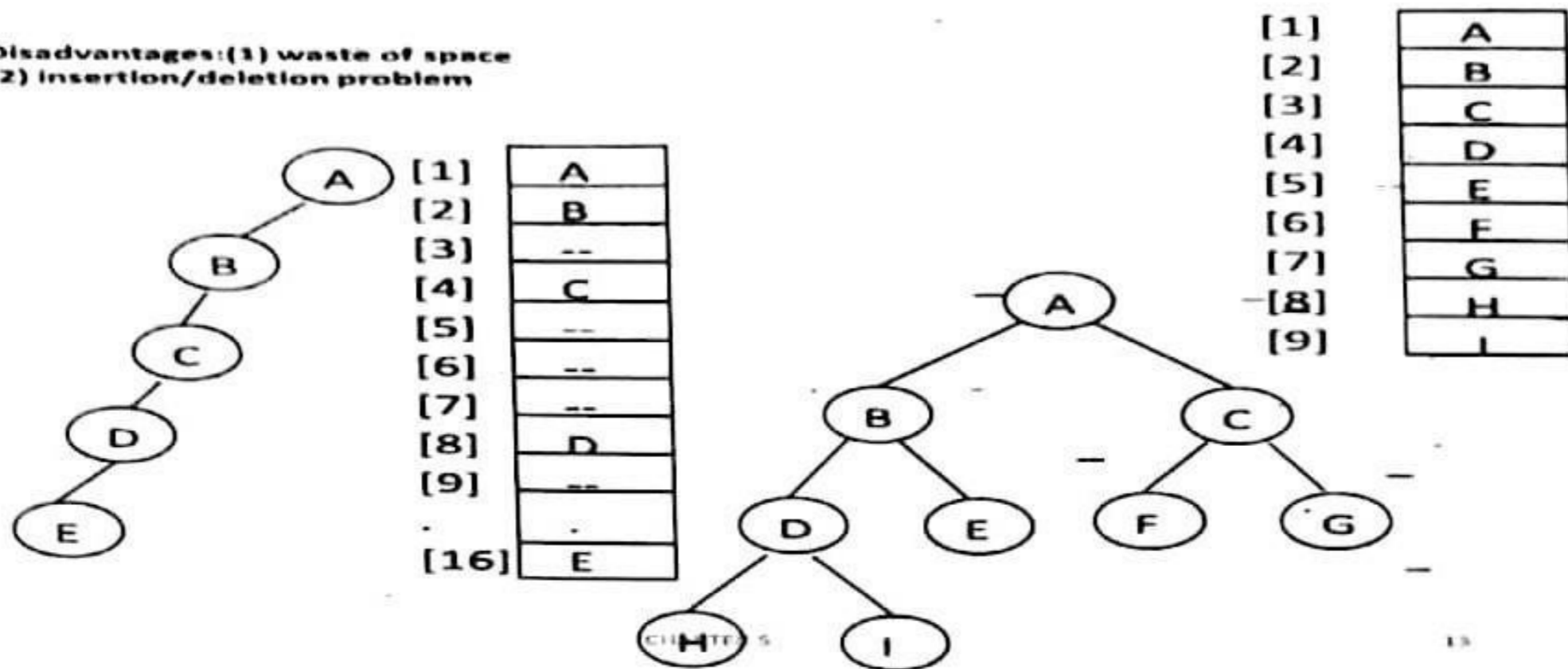
Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are 1) Array Representation 2) Linked List Representation

1) Array Representation: In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of

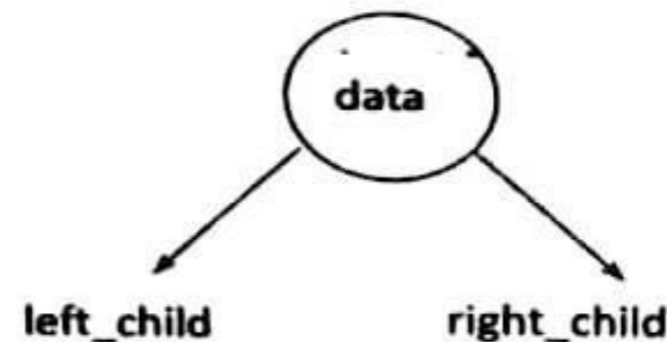
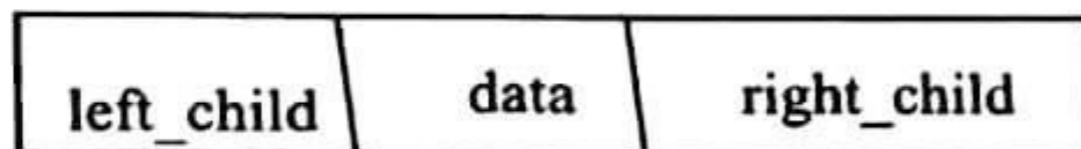
A complete binary tree with n nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have: a) $parent(i)$ is at $i/2$ if $i \neq 1$. If $i=1$, i is at the root and has no parent. b) $left_child(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child. c) $right_child(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.

Disadvantages: (1) waste of space
(2) insertion/deletion problem

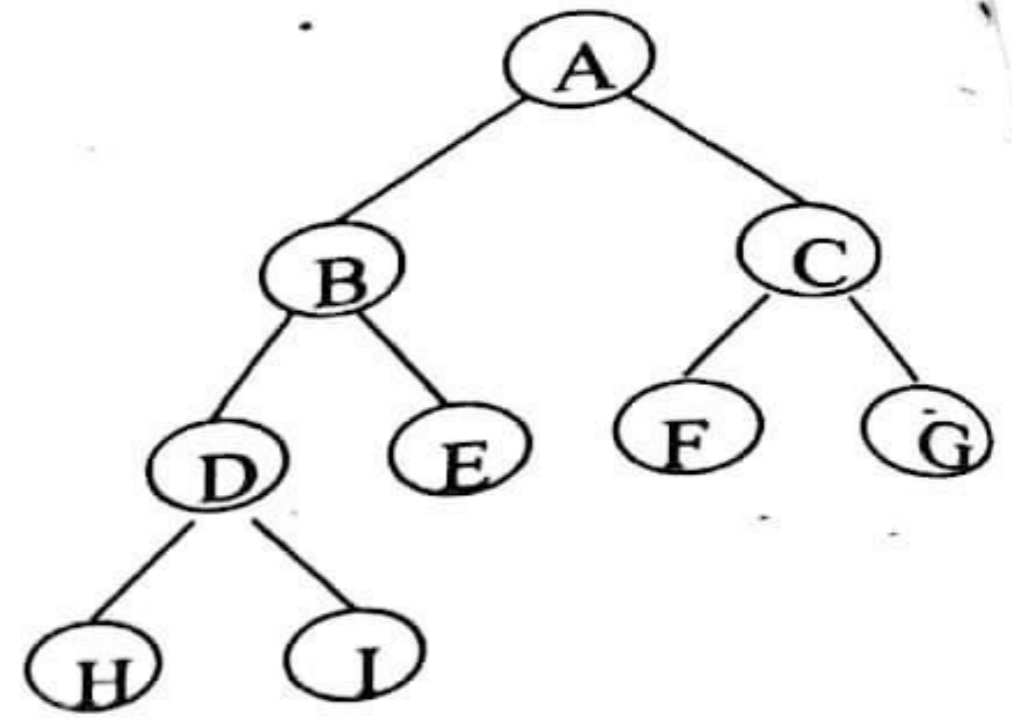
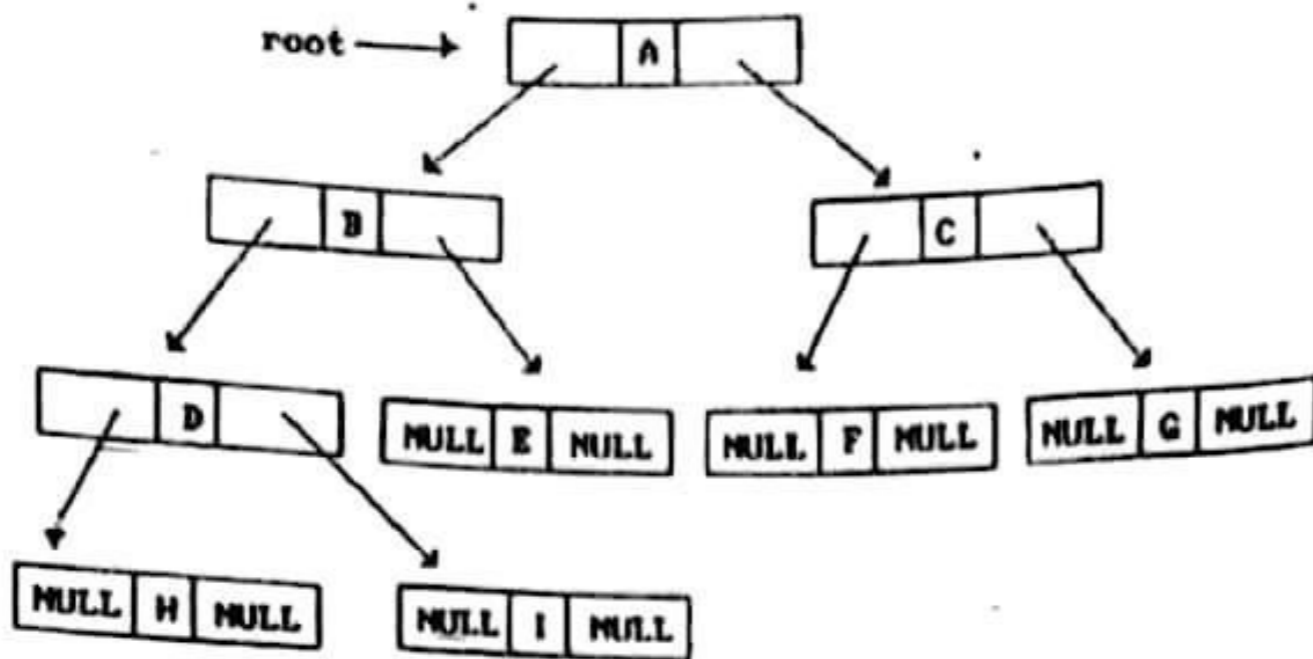


2. Linked Representation

We use linked list to represent a binary tree. In a linked list, every node consists of three fields. First field, for storing left child-address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...



```
typedef struct node *tree_pointer;
typedef struct node {
    int data;
    tree_pointer left_child, right_child;};
```



Binary Tree Traversals

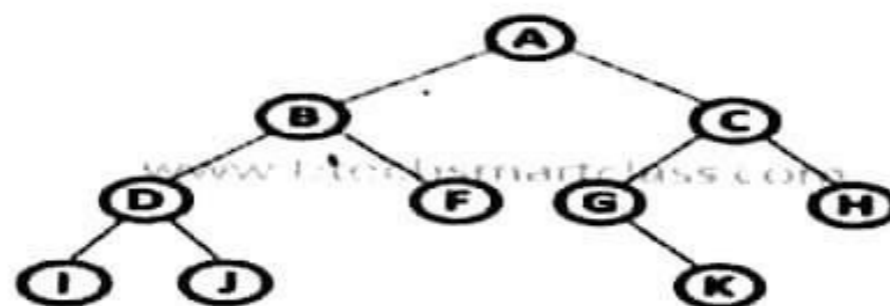
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

- 1) In - Order Traversal 2) Pre - Order Traversal 3) Post - Order Traversal

Binary Tree Traversals

- 1. In - Order Traversal (leftChild - root - rightChild)
- I - D - J - B - F - A - G - K - C - H
- 2. Pre - Order Traversal (root - leftChild - rightChild)
- A - B - D - I - J - F - C - G - K - H
- 3. Post - Order Traversal (leftChild - rightChild - root)
- I - J - D - F - B - K - G - H - C - A



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we

visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

Algorithm

Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.

```
void inorder(tree_pointer ptr) - /* inorder tree traversal */ Recursive
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
    }
}
```

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root

UNIT- IV

for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right-most child in the tree. So we stop the process. That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Algorithm

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

```
void preorder(tree_pointer ptr)    /* preorder tree traversal */    Recursive
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        preorder(ptr->right_child);
    }
}
```

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Algorithm

Until all nodes are traversed –

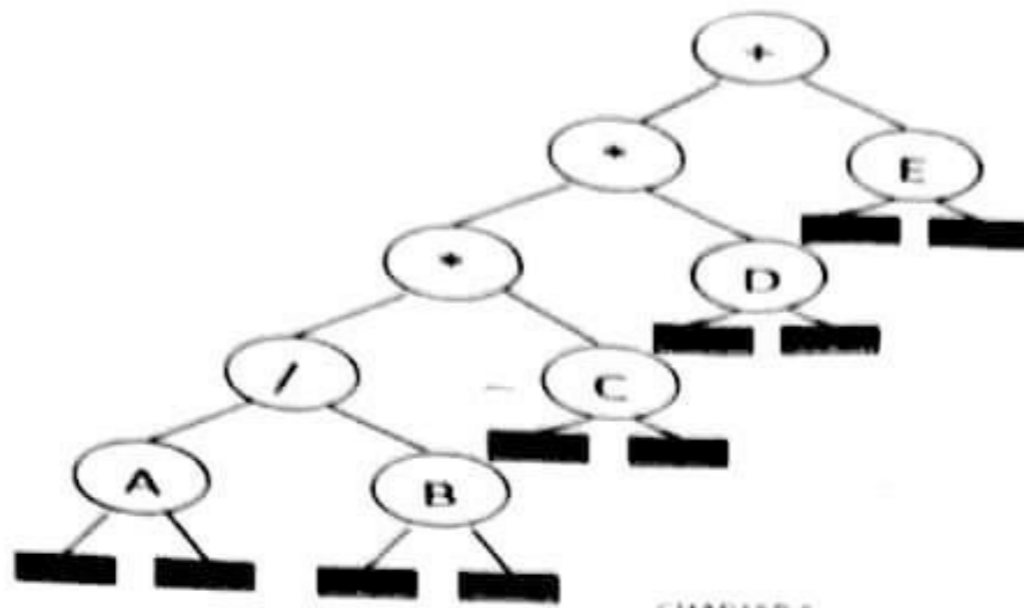
Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

```
void postorder(tree_pointer ptr)    /* postorder tree traversal */    Recursive
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

Arithmetic Expression Using BT



inorder traversal
 A / B * C * D + E
 infix expression
 preorder traversal
 + * * / A B C D E
 prefix expression
 postorder traversal
 A B / C * D * E +
 postfix expression
 level order traversal
 + * E * D / C A B

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Iterative Inorder Traversal (using stack)

```
void iter_inorder(tree_pointer node)
{
    int top = -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node = node->left_child)
            add(&top, node); /* add to stack */
        node = delete(&top); /* delete from stack */
        if (!node) break; /* empty stack */
        printf("%D", node->data);
        node = node->right_child;
    }
}
```

In iterative inorder traversal, we must create our own stack to add and remove nodes as in recursion.—

Analysis: Let n be number of nodes in tree, every node of tree is placed on and removed from the stack exactly once.

So time complexity is O(n). The space requirement is equal to the depth of tree which is O(n).

Level Order Traversal (Using Queue) – Traversal without Stack

```

void level_order(tree_pointer ptr)      /* level order tree traversal */
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty queue */
    addq(front, &rear, ptr);
    for (;;) {
        ptr = deleteq(&front, rear);
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->left_child)
                addq(front, &rear, ptr->left_child);
            if (ptr->right_child)
                addq(front, &rear, ptr->right_child);
        }
        else break;
    }
}

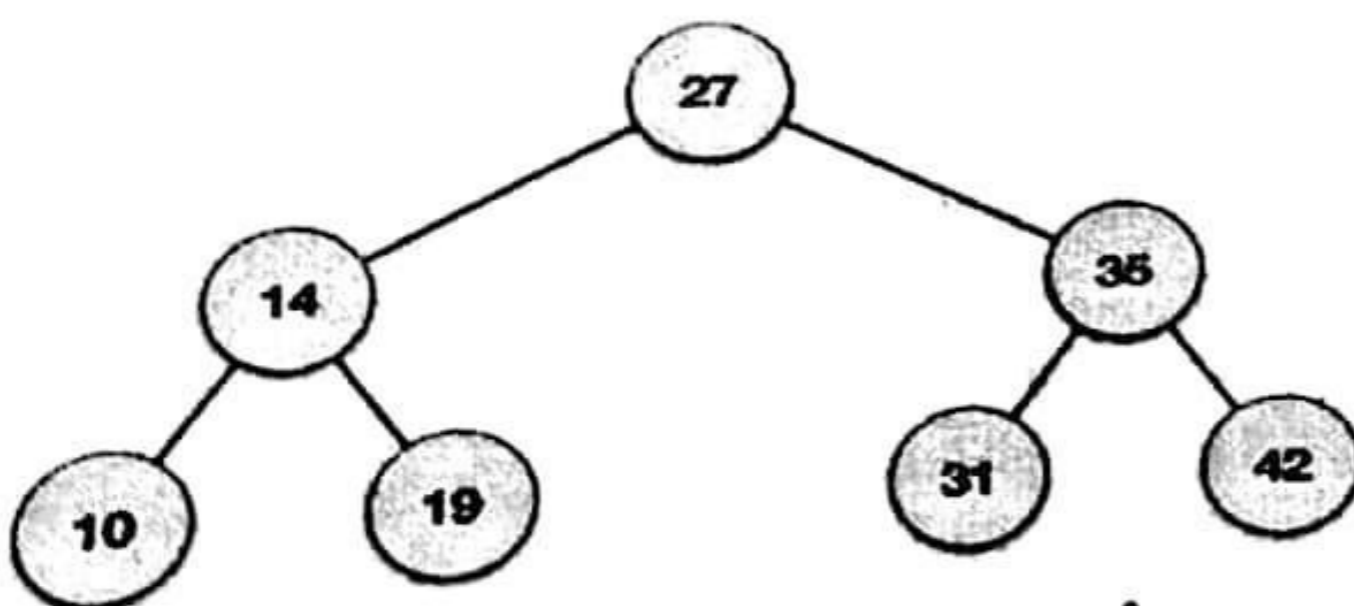
```

Level order Traversal is implemented with circular queue. In this order, we visit the root first, then root's left child followed by root's right child. We continue in this manner, visiting the nodes at each new level from left most to right most nodes.

We begin by adding root to the queue. The function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. The level order traversal for above arithmetic expression is + * E * D / C A B.

Binary Search Trees**Binary Search Tree Representation**

Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty, then all nodes follows the below mentioned properties –

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

left sub-tree and right sub-tree and can be defined as –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

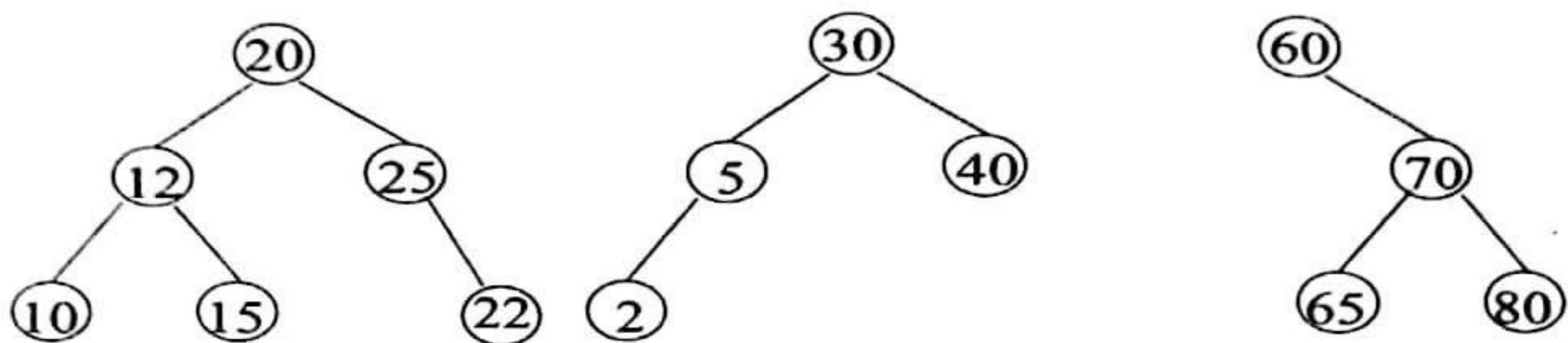


Fig: Example Binary Search Trees

ADT for Dictionary:

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following –

- **Search** – search an element in a binary search tree.
- **Insert** – insert an element into a binary search tree / create a tree.
- **Delete** – Delete an element from a binary search tree.
- **Height** -- Height of a binary search tree.

Searching a Binary Search Tree

Let an element k is to search in binary search tree. Start search from root node of the search tree. If root is NULL, search tree contains no nodes and search unsuccessful. Otherwise, compare k with the key in the root. If k equals the root's key, terminate search, if k is less than key value, search

UNIT- IV

element k in left subtree otherwise search element k in right subtree. The function search recursively searches the subtrees.

Algorithm: Recursive search of a Binary Search Tree

```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key. If there is no such
node, return NULL */
if (!root) return NULL;
if (key == root->data) return root;
if (key < root->data)
return search(root->left_child, key);
return search(root->right_child, key);
}
```

Algorithm: Iterative search of a Binary Search Tree

```
tree_pointer search2(tree_pointer tree, int key)
{
while (tree) {
if (key == tree->data) return tree;
if (key < tree->data)
tree = tree->left_child;
else tree = tree->right_child;
}
return NULL;
}
```

Analysis of Recursive search and Iterative Search Algorithms:

If h is the height of the binary search tree, both algorithms perform search in $O(h)$ time. Recursive search requires additional stack space which is $O(h)$.

Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.

In a binary search tree, the insertion operation is performed with $O(\log n)$ time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

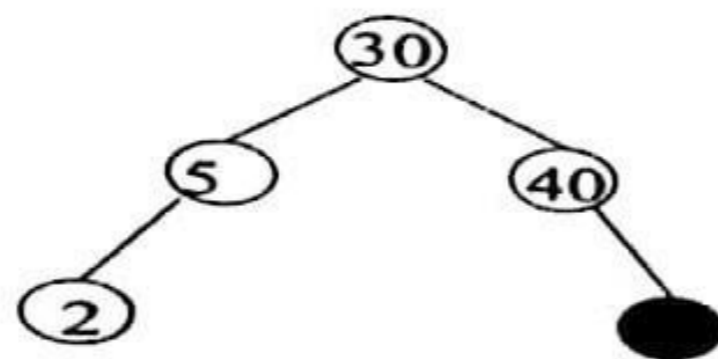
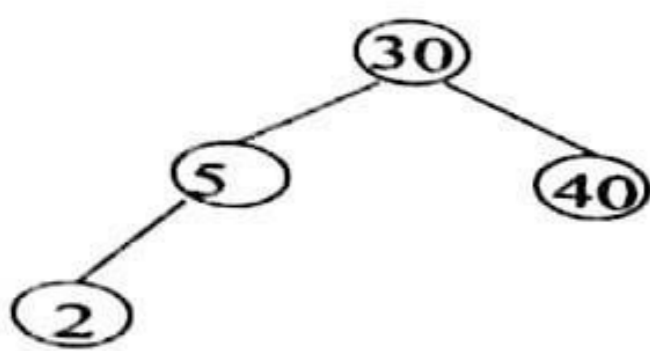
Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

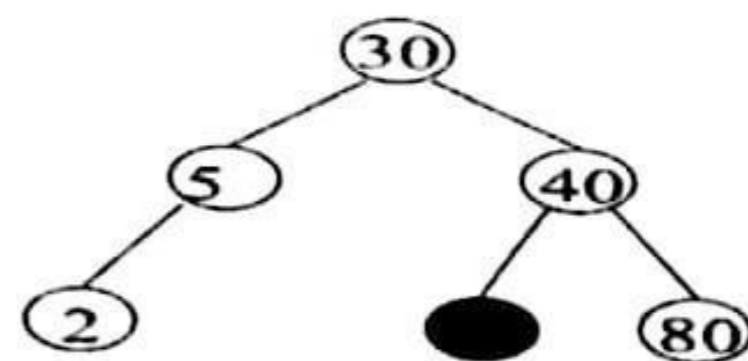
Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach a node (e.i., reach to NULL) where search terminates.

Step 7: After reaching a last node, then insert the newNode as left child if newNode is smaller or equal to that node else insert it as right child.



Insert 80



Insert 35

Algorithm

Create newnode

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

UNIT- IV

```

    goto right subtree
else
    goto left subtree
endwhile
insert newnode
end If

```

Implementation

The implementation of insert function should look like this -

```

void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;
    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;
    //if tree is empty, create root node
    if(root == NULL) {
        root = tempNode;
    }else {
        current = root;
        parent = NULL;
        while(1) {
            parent = current;
            //go to left of the tree
            if(data < parent->data) {
                current = current->leftChild;

                //insert to the left
                if(current == NULL) {
                    parent->leftChild = tempNode;

```

```

return;    }
}

//go to right of the tree
else {
    current = current->rightChild;
    //insert to the right
    if(current == NULL) {
        parent->rightChild = tempNode;
        return;
    }
}
}
}
}

```

Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, it can be divided into two stages:

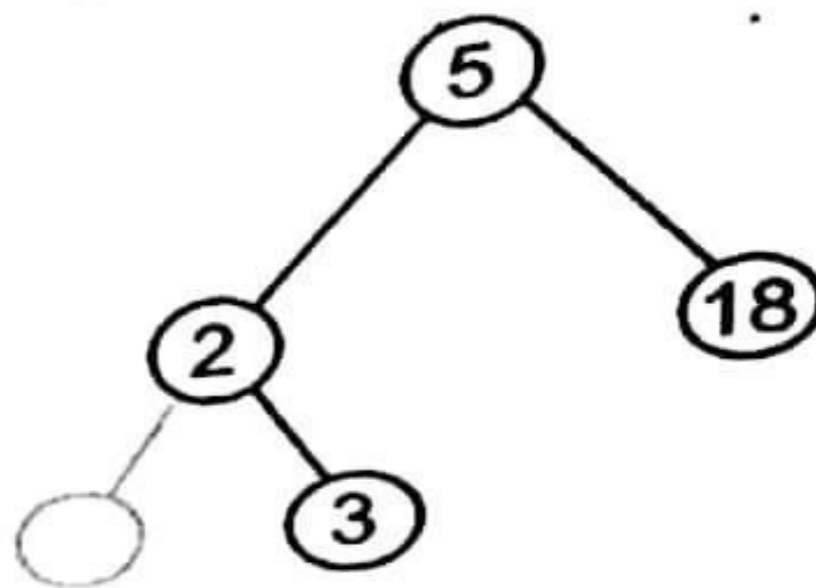
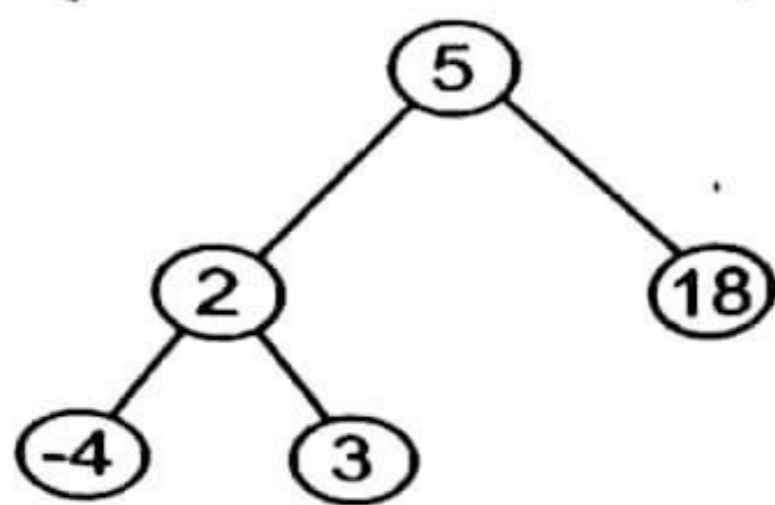
- search for a node to remove
- if the node is found, run remove algorithm.

Remove algorithm in detail

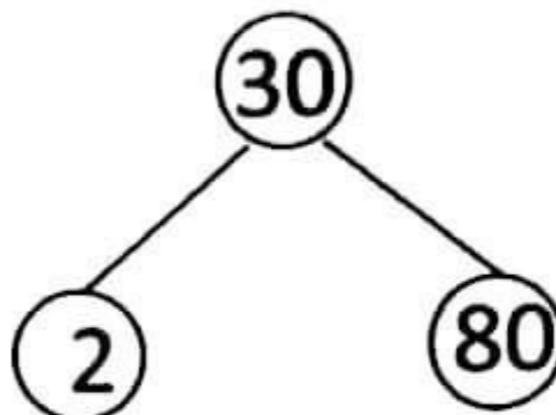
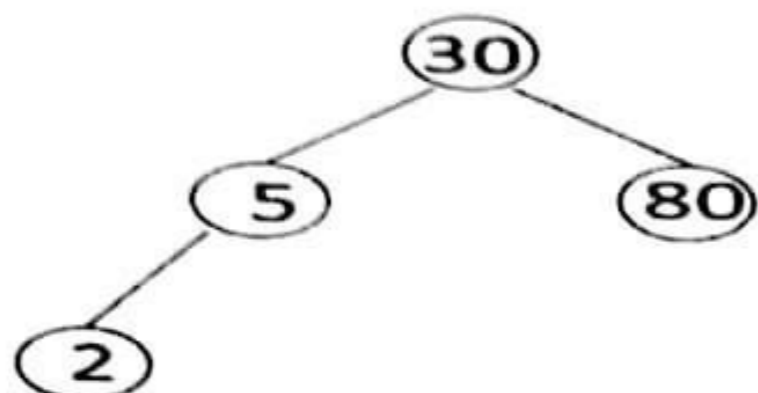
Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1. Node to be removed has no children. --This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

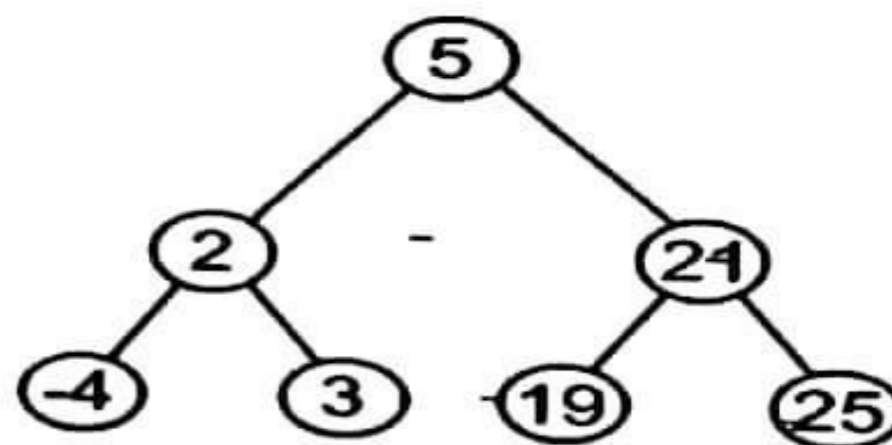
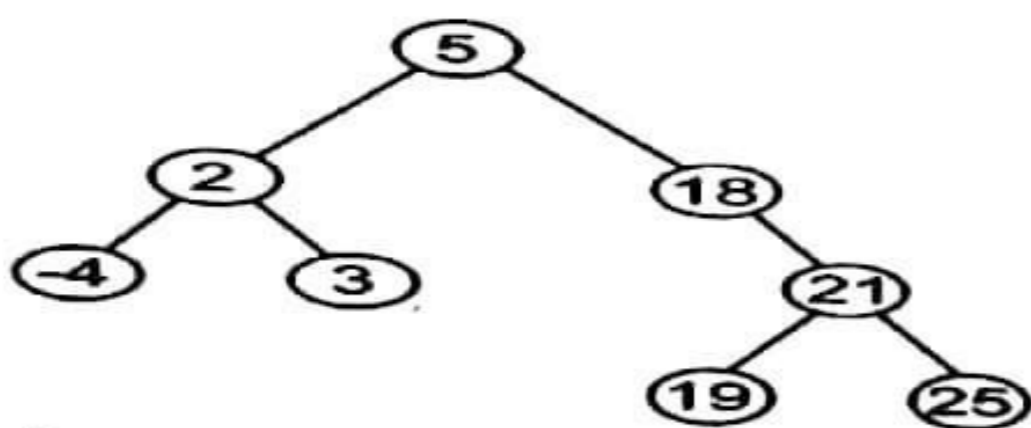
- **Example.** Remove -4 from a BST.



2. Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with its subtree) directly to the parent of the removed node.



3. Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one child.



Deletion Operation in BST

In a binary search tree, the deletion operation is performed with $O(\log n)$ time complexity. Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

```

/* deletion in binary search tree */
void deletion(struct treeNode **node, struct treeNode **parent, int data) {
    struct treeNode *tmpNode, *tmpParent;
    if (*node == NULL)
        return;
    if ((*node)->data == data) {
        /* deleting the leaf node */
        if (!(*node)->left && !(*node)->right) {
            if (parent) {
                /* delete leaf node */
            }
        }
    }
}

```

```

        if ((*parent)->left == *node)
            (*parent)->left = NULL;
        else
            (*parent)->right = NULL;
            free(*node);
    } else {
        /* delete root node with no children */
        free(*node);
    }
/* deleting node with one child */
} else if (!(*node)->right && (*node)->left) {
    /* deleting node with left child alone */
    tmpNode = *node;
    (*parent)->right = (*node)->left;
    free(tmpNode);
    *node = (*parent)->right;
} else if ((*node)->right && !(*node)->left) {
    /* deleting node with right child alone */
    tmpNode = *node;
    (*parent)->left = (*node)->right;
    free(tmpNode);
    (*node) = (*parent)->left;
} else if (!(*node)->right->left) {
    /*
     * deleting a node whose right child
     * is the smallest node in the right
     * subtree for the node to be deleted.
     */

    tmpNode = *node;

    (*node)->right->left = (*node)->left;

    (*parent)->left = (*node)->right;
    free(tmpNode);
    *node = (*parent)->left;
} else {
    /*
     * Deleting a node with two children.
     * First, find the smallest node in
     * the right subtree. Replace the
     * smallest node with the node to be
     * deleted. Then, do proper connections
     * for the children of replaced node.
     */
    tmpNode = (*node)->right;
    while (tmpNode->left) {

```



```

    tmpParent = tmpNode;
    tmpNode = tmpNode->left;
}
tmpParent->left = tmpNode->right;
tmpNode->left = (*node)->left;
tmpNode->right = (*node)->right;
free(*node);
*node = tmpNode;
}
} else if (data < (*node)->data) {
    /* traverse towards left subtree */
    deletion(&(*node)->left, node, data);
} else if (data > (*node)->data) {
    /* traversing towards right subtree */
    deletion(&(*node)->right, node, data);
}
}
}

```

Height of a Binary Search Tree:

Height of a Binary Tree For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of -1.

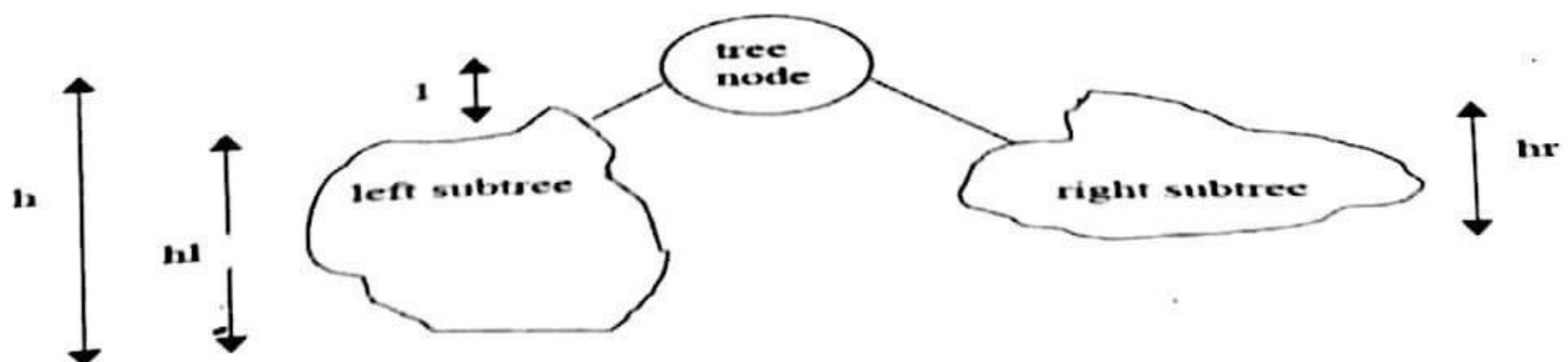
The following height function in pseudocode is defined recursively

```

int height( BinaryTree Node t) {
    if t is a null tree
        return -1;
    hl = height( left subtree of t);
    hr = height( right subtree of t);
    h = 1 + maximum of hl and hr;

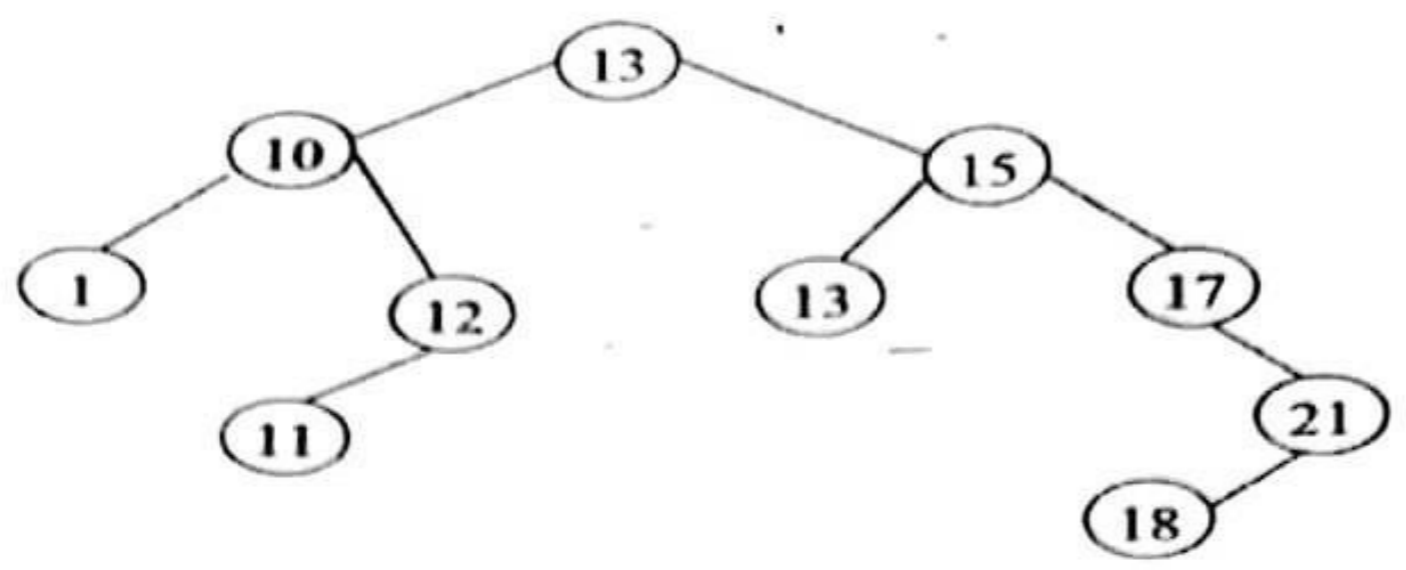
    return h;
}

```



UNIT-IV

For example, the following tree has a height of 4. Its left subtree has height 2 and its right subtree 3.



Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

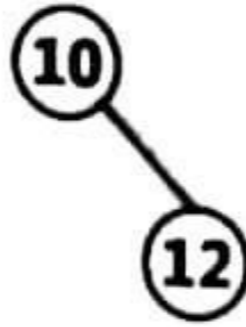
10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...

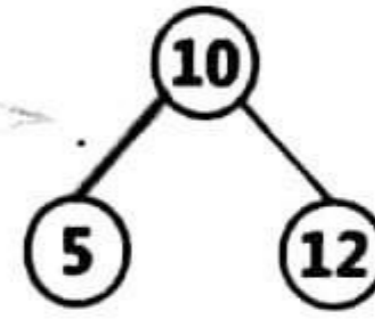
insert (10)



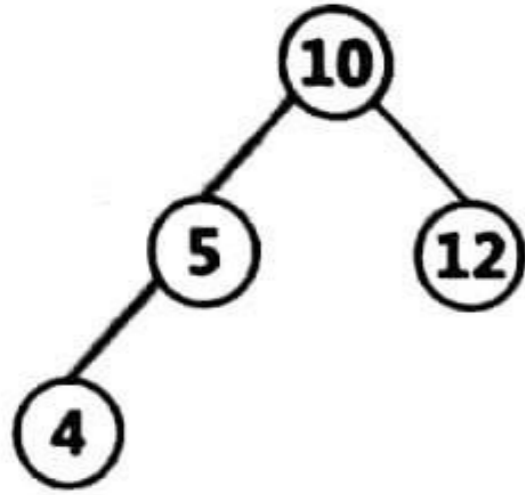
insert (12)



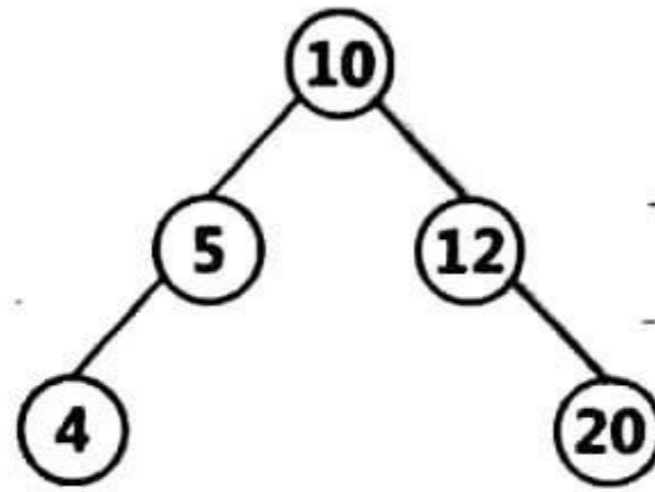
insert (5)



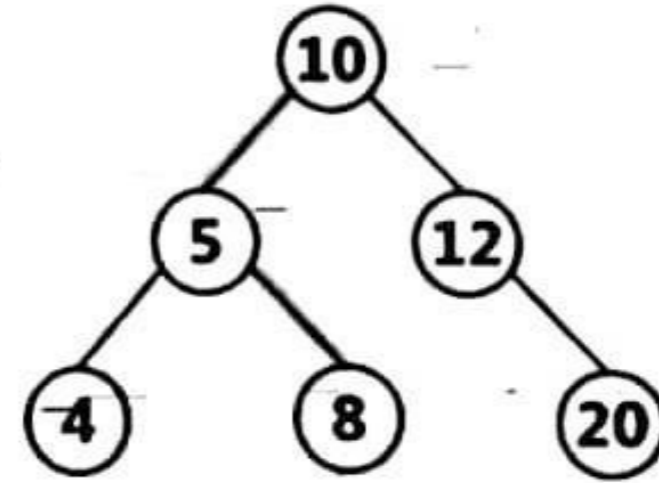
insert (4)



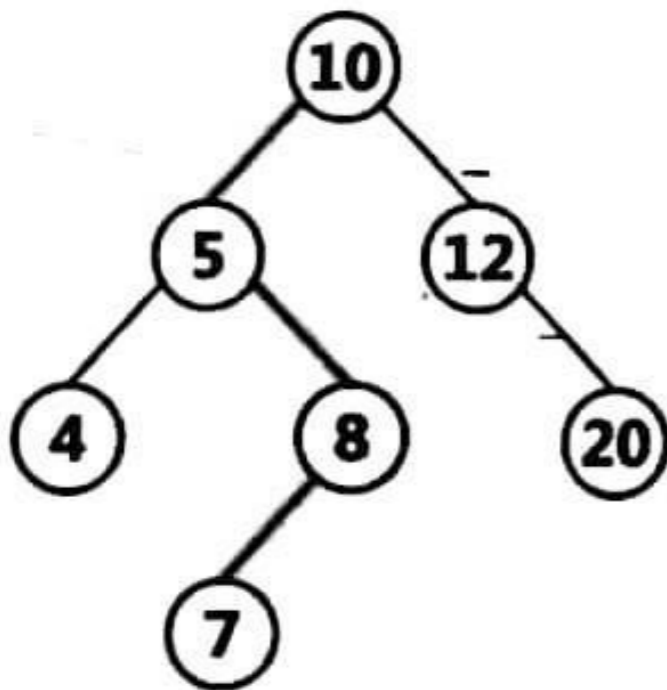
insert (20)



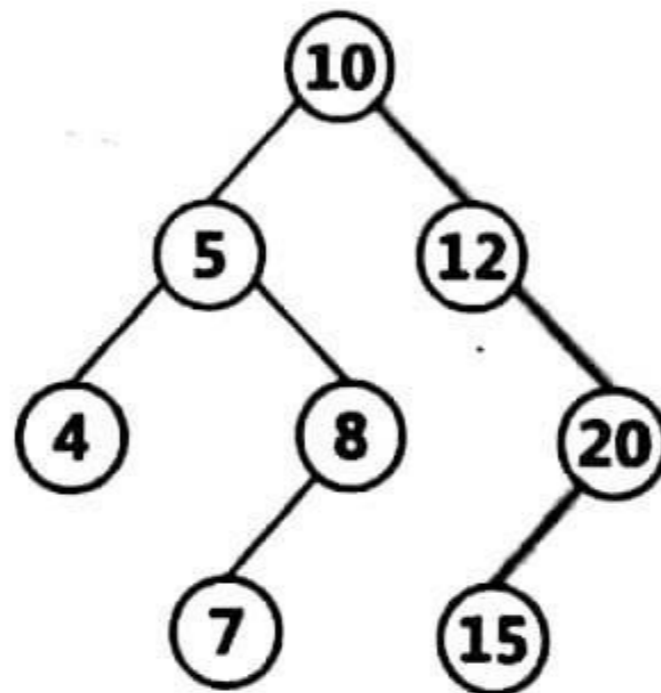
insert (8)



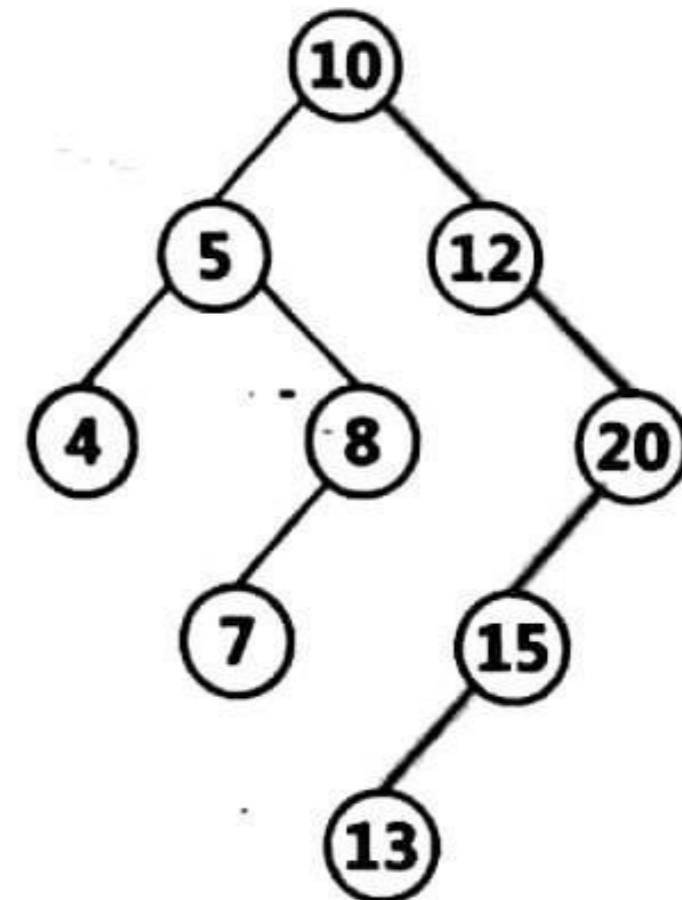
insert (7)



insert (15)



insert (13)



UNIT-V

GRAPHS

1. BASIC CONCEPTS

INTRODUCTION

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

WHY GRAPHS ARE USEFUL

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- Family trees: in which the member nodes have an edge from parent to each of their children.
- Transportation networks : in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

DEFINITION:

A graph G is defined as an ordered set (V, E) , where $V(G)$ represents the set of vertices and $E(G)$ represents the edges that connect these vertices.

We have two types of Graphs. Basically:

1. UNDIRECTED GRAPH
2. DIRECTED GRAPH

UNDIRECTED GRAPH:

Shows a graph with $V(G) = \{A, B, C, D \text{ and } E\}$ and $E(G) = \{(A, B), (B, C), (A, D), (B, D), (D,E), (C, E)\}$. Note that there are five vertices or nodes and six edges in the graph.

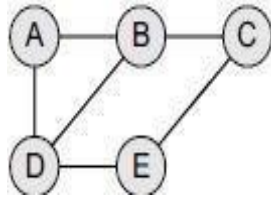


FIGURE 5.1

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 5.1 shows an undirected graph because it does not give any information about the direction of the edges.

DIRECTED GRAPH:

A directed graph G , also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G . For an edge (u, v) ,

- The edge begins at u and terminates at v .
- u is known as the origin or initial point of e . Correspondingly, v is known as the destination or terminal point of e .
- u is the predecessor of v . Correspondingly, v is the successor of u .
- Nodes u and v are adjacent to each other.

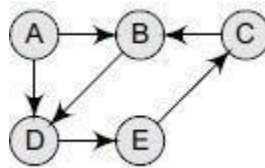


FIGURE 5.2

Which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

2. REPRESENTATION OF GRAPHS

There are two common ways of storing graphs in the computer's memory. They are:

- **Sequential representation** by using an adjacency matrix.
- **Linked representation** by using an adjacency list that stores the neighbours of a node using a linked list.

ADJACENCY MATRIX REPRESENTATION

An adjacency matrix is used to represent which nodes are adjacent to one another.

By definition: Two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G , if node v is adjacent to node u , then there is definitely an edge from u to v .

That is, if v is adjacent to u , we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of $n \times n$.

In an adjacency matrix, the rows and columns are labelled by graph vertices.

- An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other.
- However, if the nodes are not adjacent, a_{ij} will be set to zero.

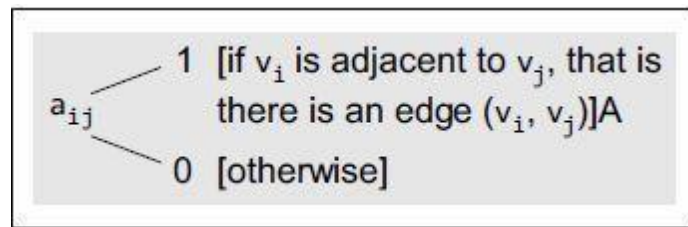


FIGURE 5.3 Adjacency Matrix Entry

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix.

The entries in the matrix depend on the ordering of the nodes in G . Therefore, a change in the order of nodes will result in a different adjacency matrix.

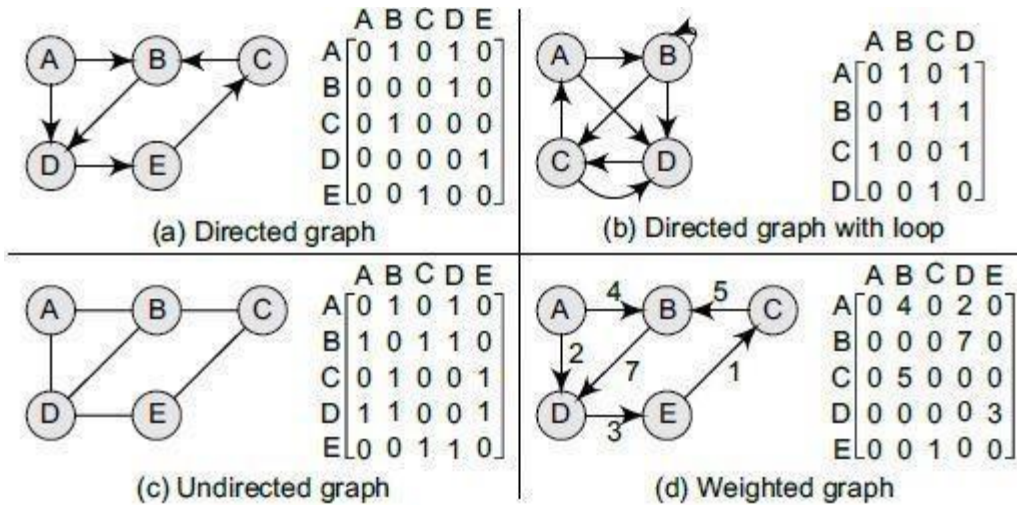


Figure 5.4 shows some graphs and their corresponding adjacency matrices.

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix:

From adjacency matrix A_1 , we can conclude that an entry 1 in the i th row and j th column means that there exists a path of length 1 from V_i to V_j . Now consider, A_2 , A_3 , and A_4 .

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (V_i, V_k) and (V_k, V_j) , then there is a path from v_i to v_j of length 2.

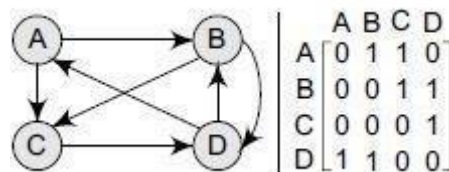


FIGURE 5.5 Directed graph with its adjacency matrix

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$$B = A^1 + A^2 + A^3 + \dots + A^r$$

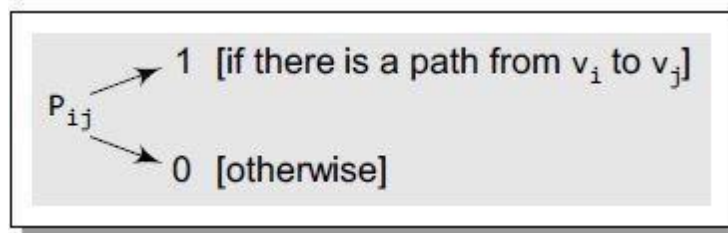


FIGURE 5.6 Path Matrix Entry

The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry $P_{ij} = 1$, if B_{ij} is non-zero and $P_{ij} = 0$, if otherwise. The path matrix is used to show whether there exists a simple path from node v_i to v_j or not.

Let us now calculate matrix B and matrix P using the above discussion.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 6 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 3 & 3 & 5 \\ 6 & 8 & 7 & 8 \end{bmatrix}$$

Now the path matrix P can be given as:

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

ADJACENCY LINKED LIST REPRESENTATION

- An adjacency list is another way in which graphs can be represented in the computer's memory.
- This structure consists of a list of all nodes in G.
- Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

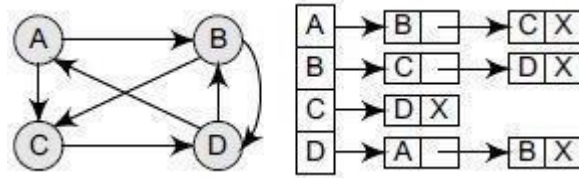


FIGURE 5.7 Graph G and its adjacency list

- For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.
- However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u .
- Adjacency lists can also be modified to store weighted graphs.

Let us now see an adjacency list for an undirected graph as well as a weighted graph.

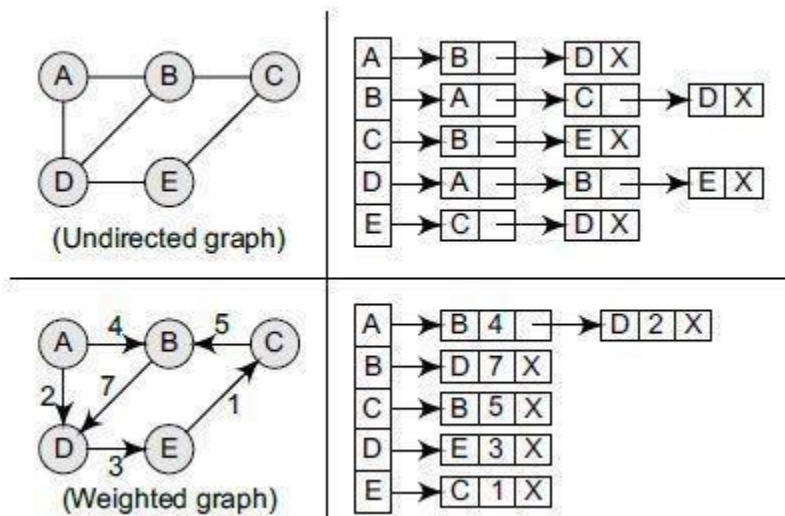


FIGURE 5.8 Adjacency list for an undirected graph and a weighted graph

PROGRAMMING EXAMPLE

1. Write a program to create a graph of n vertices using an adjacency list. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
char vertex;
struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);
int main()
{
struct node *Adj[10];
int i, no_of_nodes;
clrscr();
printf("\n Enter the number of nodes in G: ");
scanf("%d", &no_of_nodes);
for(i = 0; i < no_of_nodes; i++)
Adj[i] = NULL;
createGraph(Adj, no_of_nodes);
printf("\n The graph is: ");
displayGraph(Adj, no_of_nodes);
deleteGraph(Adj, no_of_nodes);
getch();
return 0;
}
```

```

void createGraph(struct node *Adj[], int no_of_nodes)
{
struct node *new_node, *last;
int i, j, n, val;
for(i = 0; i < no_of_nodes; i++)
{
last = NULL;
printf("\n Enter the number of neighbours of %d: ", i);
scanf("%d", &n);
for(j = 1; j <= n; j++)
{
printf("\n Enter the neighbour %d of %d: ", j, i);
scanf("%d", &val);
new_node = (struct node *) malloc(sizeof(struct node));
new_node -> vertex = val;
new_node -> next = NULL;
if (Adj[i] == NULL)
Adj[i] = new_node;
else
last -> next = new_node;
last = new_node
}
}
}
void displayGraph (struct node *Adj[], int no_of_nodes)

```

Graphs **393**

```

{
struct node *ptr;
int i;
for(i = 0; i < no_of_nodes; i++)
{

```

```

ptr = Adj[i];
printf("\n The neighbours of node %d are:", i);
while(ptr != NULL)
{
printf("\t%d", ptr -> vertex);
ptr = ptr -> next;
}
}
}

void deleteGraph (struct node *Adj[], int no_of_nodes)
{
int i;
struct node *temp, *ptr;
for(i = 0; i <= no_of_nodes; i++)
{
ptr = Adj[i];
while(ptr != NULL)
{
temp = ptr;
ptr = ptr -> next;
free(temp);
}
Adj[i] = NULL;
}
}

```

Output

```

Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 2
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0

```

Enter the neighbour 2 of 1: 2

Enter the number of neighbours of 2: 1

Enter the neighbour 1 of 2: 1

The neighbours of node 0 are: 1

The neighbours of node 1 are: 0 2

The neighbours of node 2 are: 0

Note If the graph in the above program had been a weighted graph, then the structure of the node would have been:

```
typedef struct node
{
int vertex;
int weight;
struct node *next;
};
```

3. GRAPH TRAVERSALS

There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search
2. Depth-first search

1. Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

ALGORITHM

Step 1: SET STATUS = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node A

and set its STATUS = 2

(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).

Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)

[END OF LOOP]

Step 6: EXIT

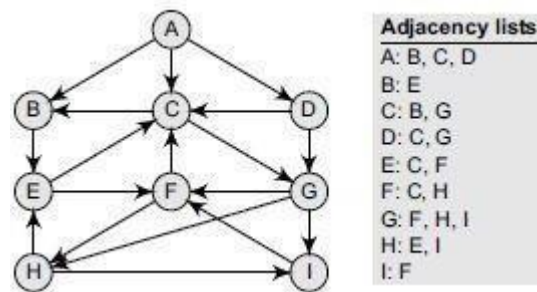


FIGURE 5.9 Graph G And Its Adjacency List

EXAMPLE

Consider the graph G given in Fig. 5.9. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

SOLUTION:

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:

1. QUEUE
 2. ORIG
- While QUEUE is used to hold the nodes that have to be processed,
 - ORIG is used to keep track of the origin of each edge.
 - Initially, FRONT = REAR = -1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0 QUEUE = A

REAR = 0 ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1 QUEUE = A B C D

REAR = 3 ORIG = \0 A A A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2 QUEUE = A B C D E

REAR = 4 ORIG = \0 A A A B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5 QUEUE = A B C D E G F

REAR = 6 ORIG = \0 A A A B C E

(g) Dequeue a node by setting $FRONT = FRONT + 1$ and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6 QUEUE = A B C D E G F H I

REAR = 9 ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have

P as A -> C -> G -> I.

Features of Breadth-First Search Algorithm

Space complexity:

The space complexity is therefore proportional to the number of nodes at the deepest level of the graph.

Given a graph with branching factor b (number of children at each node) and depth d , the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$.

The space complexity can also be expressed as $O(|E| + |V|)$, where $|E|$ is the total number of edges in G and $|V|$ is the number of nodes or vertices.

Time Complexity:

In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(b^d)$.

However, the time complexity can also be expressed as $O(|E| + |V|)$, since every vertex and every edge will be explored in the worst case.

Completeness:

Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality:

Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node.

we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Programming Example

2. Write a program to implement the breadth-first search algorithm.

```
#include <stdio.h>

#define MAX 10

void breadth_first_search(int adj[][MAX],int visited[],int start)
{
int queue[MAX],rear = -1,front = -1, i;

queue[++rear] = start;

visited[start] = 1;

while(rear != front)
{
start = queue[++front];

if(start == 4)

printf("5\t");

else

printf("%c \t",start + 65);

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

```

if(adj[start][i] == 1 && visited[i] == 0)
{
queue[++rear] = i;
visited[i] = 1;
}
}
}
}

int main()
{
int visited[MAX] = {0};
int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
return 0;
}

```

Output

Enter the adjacency matrix:

0 1 0 1 0

1 0 1 1 0

0 1 0 0 1

1 1 0 0 1

0 0 1 1 0

A B D C E

2. Depth First Algorithm

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.

When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set
its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N . Process it and set its
STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that
are in the ready state (whose STATUS = 1) and
set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

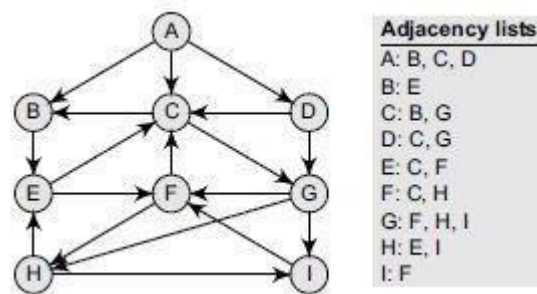


FIGURE 5.10 Graph G And Its Adjacency List

Example:

Consider the graph G given in. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H . The procedure can be explained here.

Solution:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity:

The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity:

The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as ($O(|V|+|E|)$).

Completeness:

Depth-first search is said to be a complete algorithm. If there is a solution, depthfirst search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Programming Example:

3. Write a program to implement the depth-first search algorithm.

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
int stack[MAX];
int top = -1, i;
printf("%c-",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top != -1)
{
start = stack[top];
for(i = 0; i < MAX; i++)
{
if(adj[start][i] && visited[i] == 0)
{
stack[++top] = i;
printf("%c-", i + 65);
visited[i] = 1;
break;
}
}
if(i == MAX)
top--;
}
}
int main()
{
int adj[MAX][MAX];
```

```

int visited[MAX] = {0}, i, j;
400 Data Structures Using C
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
printf("DFS Traversal: ");
depth_first_search(adj,visited,0);
printf("\n");
return 0;
}

```

Output

Enter the adjacency matrix:

0 1 0 1 0

1 0 1 1 0

0 1 0 0 1

1 1 0 0 1

0 0 1 1 0

DFS Traversal: A -> C -> E ->

APPLICATIONS

MINIMUM SPANNING TREES:

- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together
- A graph G can have many different spanning trees.
- We can assign *weights* to each edge (which is a number that represents how unfavourable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.
- A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning

tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Example: Consider an unweighted graph G given below (Fig. 5.11). From G , we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.

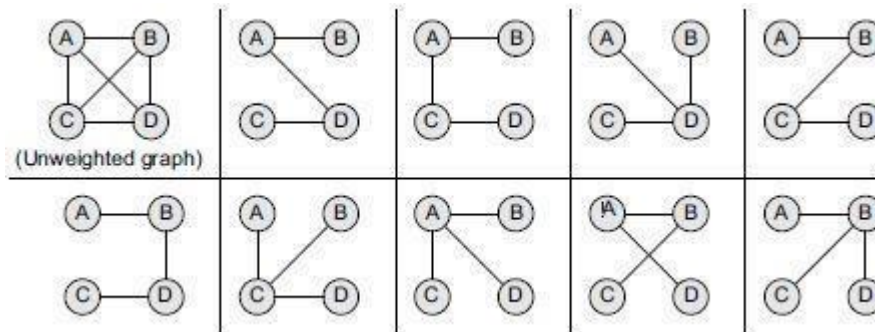


FIGURE 5.11 Unweighted Graph And Its Spanning Trees

EXAMPLE: Consider a weighted graph G shown in Fig. 5.12. From G , we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. Of all the spanning trees given in Fig. 5.12, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.

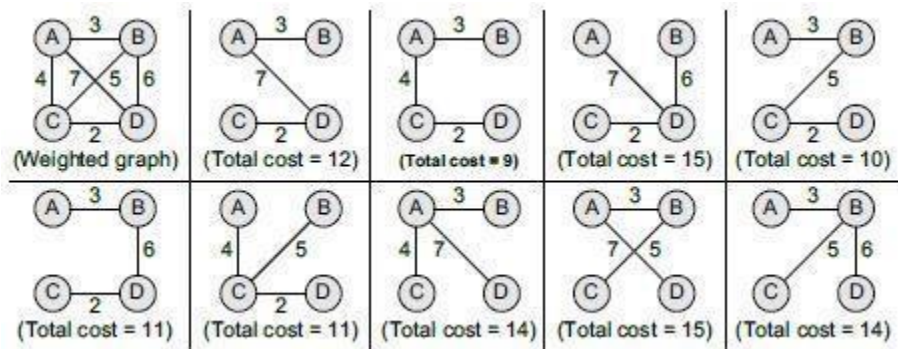


FIGURE 5.12 Weighted Graph And Its Spanning Trees.

APPLICATIONS FOR MINIMUM SPANNING TREES:

- **MST'S** is widely used for designing networks.
- **MST'S** are used to find airline routes.
- **MST'S** are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- **MST'S** are applied in routing algorithms for finding the most efficient path.

We have two types of ALGORITHMS in Minimum Spanning Trees. They are:

1. PRIM'S ALGORITHM
2. KRUSKAL'S ALGORITHM

1. PRIM'S ALGORITHM

- Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.
- In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized.

For this, the algorithm maintains three sets of vertices which can be given as below:

- **Tree vertices** Vertices that are a part of the minimum spanning tree T.
- **Fringe vertices** Vertices that are currently not a part of T, but are adjacent to some tree vertex.
- **Unseen vertices** Vertices that are neither tree vertices nor fringe vertices fall under this category.

ALGORITHM

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and
fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the
minimum spanning tree T

[END OF LOOP]

Step 5: EXIT

EXAMPLE: Construct a minimum spanning tree of the graph given in Fig. 5.13

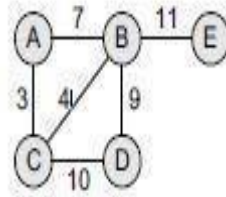


FIGURE 5.13

Step 1: Choose a starting vertex A.

Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.

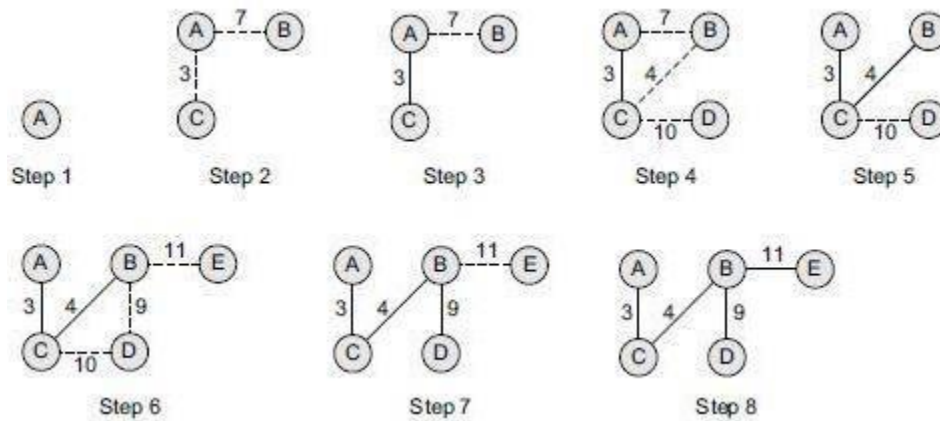
Step 4: Add the fringe vertices (that are adjacent to C).

Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.

Step 6: Add the fringe vertices (that are adjacent to B).

Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.

Step 8: Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n-1 edges that have minimum weight. So, the minimum spanning tree can now be given as,



2. KRUSKAL'S ALGORITHM

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized.
- However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

ALGORITHM

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees,
 then Add it to the forest (for combining two trees into one
 tree).

ELSE

Discard the edge

Step 6: END

EXAMPLE: Apply Kruskal's algorithm on the graph given in Fig. 5.14.

Initially, we have $F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\}$

$MST = \{\}$

$Q = \{(A, D), (E, F), (C, E), (E, D), (C, D), (D, F),$
 $(A, C), (A, B), (B, C)\}$

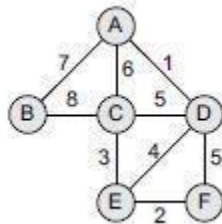
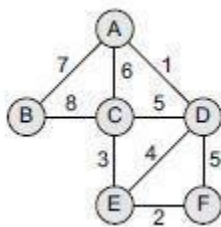


FIGURE 5.14

Step 1: Remove the edge (A, D) from Q and make the following changes:

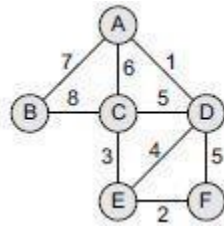


$F = \{\{A, D\}, \{B\}, \{C\}, \{E\}, \{F\}\}$

$MST = \{A, D\}$

$Q = \{(E, F), (C, E), (E, D), (C, D),$
 $(D, F), (A, C), (A, B), (B, C)\}$

Step 2: Remove the edge (E, F) from Q and make the following changes:

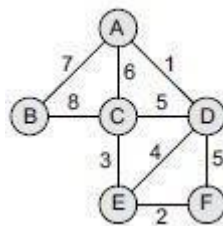


$$F = \{\{A, D\}, \{B\}, \{C\}, \{E, F\}\}$$

$$\text{MST} = \{(A, D), (E, F)\}$$

$$Q = \{(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)\}$$

Step 3: Remove the edge (C, E) from Q and make the following changes:

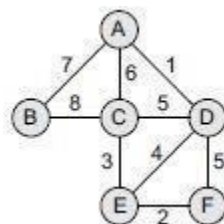


$$F = \{\{A, D\}, \{B\}, \{C, E, F\}\}$$

$$\text{MST} = \{(A, D), (C, E), (E, F)\}$$

$$Q = \{(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)\}$$

Step 4: Remove the edge (E, D) from Q and make the following changes:



$$F = \{\{A, C, D, E, F\}, \{B\}\}$$

$$\text{MST} = \{(A, D), (C, E), (E, F), (E, D)\}$$

$$Q = \{(C, D), (D, F), (A, C), (A, B), (B, C)\}$$

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$

$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$

$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$

$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$

$$Q = \{(A, C), (A, B), (B, C)\}$$

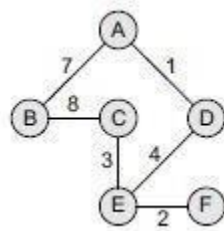
Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$

$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$

$$Q = \{(A, B), (B, C)\}$$

Step 8: Remove the edge (A, B) from Q and make the following changes:

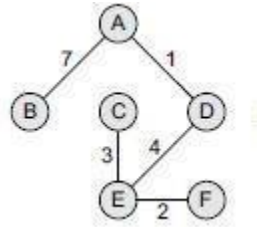


$$F = \{A, B, C, D, E, F\}$$

$$MST = \{(A, D), (C, E), (E, F), (E, D), (A, B)\}$$

$$Q = \{(B, C)\}$$

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below



$F = \{A, B, C, D, E, F\}$

$MST = \{(A, D), (C, E), (E, F), (E, D), (A, B)\}$

$Q = \{\}$

PROGRAMMING EXAMPLE:

5. Write a program which finds the cost of a minimum spanning tree.

```

#include<stdio.h>
#include<conio.h>
#define MAX 10
int adj[MAX][MAX], tree[MAX][MAX], n;
void readmatrix()
{
int i, j;
printf("\n Enter the number of nodes in the Graph : ");
scanf("%d", &n);
printf("\n Enter the adjacency matrix of the Graph");
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
scanf("%d", &adj[i][j]);
}
int spanningtree(int src)
{

```



```

int visited[MAX], d[MAX], parent[MAX];
int i, j, k, min, u, v, cost;
for (i = 1; i <= n; i++)
{
d[i] = adj[src][i];
visited[i] = 0;
parent[i] = src;
}
visited[src] = 1;
cost = 0;
k = 1;
for (i = 1; i < n; i++)
{
min = 9999;
for (j = 1; j <= n; j++)
{
if (visited[j]==0 && d[j] < min)
{
min = d[j];
u = j;
cost += d[u];
}
}
visited[u] = 1;
//cost = cost + d[u];
tree[k][1] = parent[u];
tree[k++][2] = u;
for (v = 1; v <= n; v++)
if (visited[v]==0 && (adj[u][v] < d[v]))
{
d[v] = adj[u][v];

```

```
parent[v] = u;
}
}
return cost;
}
void display(int cost)
{
int i;
printf("\n The Edges of the Mininum Spanning Tree are");
for (i = 1; i < n; i++)
printf(" %d %d \n", tree[i][1], tree[i][2]);
printf("\n The Total cost of the Minimum Spanning Tree is : %d", cost);
}
main()
{
int source, treecost;
readmatrix();
printf("\n Enter the Source : ");
scanf("%d", &source);
treecost = spanningtree(source);
display(treecost);
return 0;
}
```

Output

Enter the number of nodes in the Graph : 4

Enter the adjacency matrix : 0 1 1 0

0 0 0 1

0 1 0 0

1 0 1 0

Enter the source : 1

The edges of the Minimum Spanning Tree are 1 4

4 2

2 3

The total cost of the Minimum Spanning Tree is : 1

Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Given a graph G and a source node A , the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

ALGORITHM

- Dijkstra's algorithm is used to find the length of an *optimal* path between two nodes in a graph.
- The term *optimal* can mean anything, shortest, cheapest, or fastest.
- If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node.

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with ∞ , and insert it into N .
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N .
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.
(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in N that has the smallest label assigned to it and add it to N .

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node.

There are two kinds of labels: **temporary** and **permanent**.

Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labelled, then there is no path from the source to the destination node.

EXAMPLE:

Consider the graph G given in Fig. 5.14. Taking D as the initial node, execute the Dijkstra's algorithm on it.

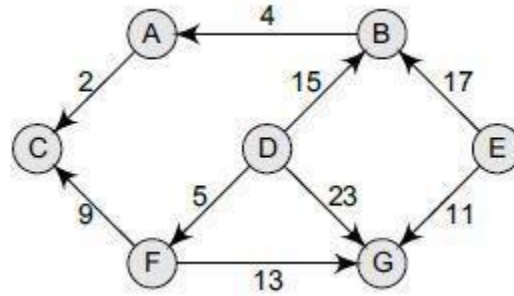


FIGURE 5.14

Step 1: Set the label of $D = 0$ and $N = \{D\}$.

Step 2: Label of $D = 0$, $B = 15$, $G = 23$, and $F = 5$. Therefore, $N = \{D, F\}$.

Step 3: Label of $D = 0$, $B = 15$, G has been re-labelled 18 because minimum $(5 + 13, 23) = 18$, C has been re-labelled 14 $(5 + 9)$. Therefore, $N = \{D, F, C\}$.

Step 4: Label of $D = 0$, $B = 15$, $G = 18$. Therefore, $N = \{D, F, C, B\}$.

Step 5: Label of $D = 0$, $B = 15$, $G = 18$ and $A = 19$ $(15 + 4)$. Therefore, $N = \{D, F, C, B, G\}$.

Step 6: Label of $D = 0$ and $A = 19$. Therefore, $N = \{D, F, C, B, G, A\}$

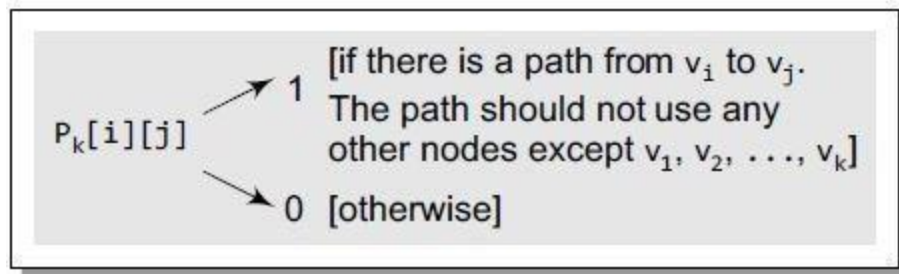
Note that we have no labels for node E ; this means that E is not reachable from D . Only the nodes that are in N are reachable from D .

The running time of Dijkstra's algorithm can be given as $O(|V|^2 + |E|) = O(|V|^2)$ where V is the set of vertices and E in the graph.

Warshall's Algorithm

If a graph G is given as $G=(V, E)$, where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, $P = A + A^2 + A^3 + \dots + A^n$.

This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices $P_0, P_1, P_2, \dots, P_n$.



Path Matrix Entry

- This means that if $P_0[i][j] = 1$, then there exists an edge from node v_i to v_j .
- If $P_1[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1 .

Hence, the path matrix P_n can be calculated with the formula given as:

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][k] \wedge P_{k-1}[k][j])$$

where \vee indicates logical OR operation and \wedge indicates logical AND operation.

ALGORITHM

Step 1: [the Path Matrix] Repeat Step 2 for $I =$ to $n-1$,

where n is the number of nodes in the graph

Step 2: Repeat Step 3 for $J =$ to $n-1$

Step 3: IF $A[I][J] =$, then SET $P[I][J] =$

ELSE $P[I][J] = 1$

[END OF LOOP]

[END OF LOOP]

Step 4: [Calculate the path matrix P] Repeat Step 5 for $K =$ to $n-1$

Step 5: Repeat Step 6 for $I =$ to $n-1$

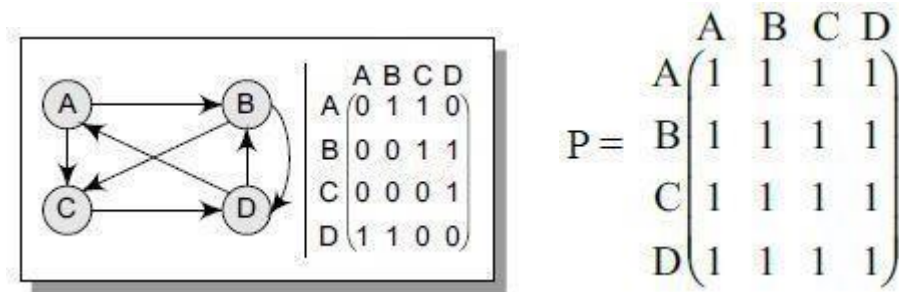
Step 6: Repeat Step 7 for J= to n-1

Step 7: SET $P [I][J] = P [I][J] \vee (P [I][K] \vee P [K][J])$

Step 8: EXIT

EXAMPLE:

Consider the graph in Fig. 13.39 and its adjacency matrix A. We can straightaway calculate the path matrix P using the Warshall's algorithm. The path matrix P can be given in a single step as:



GRAPH G AND ITS PATH MATRIX

PROGRAMMING EXAMPLE

6. Write a program to implement Warshall's algorithm to find the path matrix.

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

```
#include <conio.h>
```

```
void read (int mat[5][5], int n);
```

```
void display (int mat[5][5], int n);
```

```
void mul(int mat[5][5], int n);
```

```
int main()
```

```
{
```

```
int adj[5][5], P[5][5], n, i, j, k;
```

```
clrscr();
```

```
printf("\n Enter the number of nodes in the graph : ");
```

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

```
printf("\n Enter the adjacency matrix : ");
```

```
read(adj, n);
```

```

clrscr();
printf("\n The adjacency matrix is : ");
display(adj, n);
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(adj[i][j] == 0)
P[i][j] = 0;
else
P[i][j] = 1;
}
}
for(k=0; k<n;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
P[i][j] = P[i][j] | ( P[i][k] & P[k][j]);
}
}
printf("\n The Path Matrix is :");
display (P, n);
getch();
return 0;
}
void read(int mat[5][5], int n)
{
int i, j;
for(i=0;i<n;i++)
{

```



```

for(j=0;j<n;j++)
{
printf("\n mat[%d][%d] = ", i, j);
scanf("%d", &mat[i][j]);
}
}
}
void display(int mat[5][5], int n)
{
int i, j;
for(i=0;i<n;i++)
printf("\n");
for(j=0;j<n;j++)
printf("%d\t", mat[i][j]);
}
}

```

Output

The adjacency matrix is

0 1 1 0

0 0 1 1

0 0 0 1

1 1 0 0

Graphs 417

The Path Matrix is

1 1 1 1

1 1 1 1

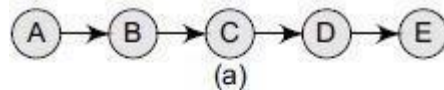
1 1 1 1

1 1 1 1

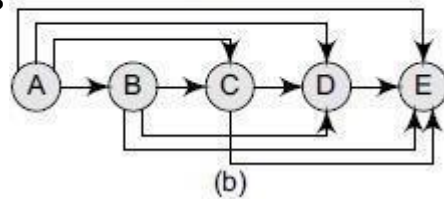
Transitive Closure of a Directed Graph

Definition

For a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph $G^* = (V, E^*)$. In G^* , for every vertex pair v, w in V there is an edge (v, w) in E^* if and only if there is a valid path from v to w in G .



Where and Why is it Needed?



(a) A graph G and its
(b) transitive closure G^*

Finding the transitive closure of a directed graph is an important problem in the following computational tasks:

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.
- It is used in the construction of parsing automata in compiler construction.
- Recently, transitive closure computation is being used to evaluate recursive database queries (because almost all practical recursive queries are transitive in nature).

ALGORITHM

Transitive_Closure(A, t, n)

Step 1: SET $i=1, j=1, k=1$

Step 2: Repeat Steps 3 and 4 while $i \leq n$

Step 3: Repeat Step 4 while $j \leq n$

Step 4: IF ($A[i][j] = 1$)

 SET $t[i][j] = 1$

 ELSE

 SET $t[i][j] =$

INCREMENT j
[END OF LOOP]
INCREMENT i
[END OF LOOP]

Step 5: Repeat Steps 6 to 11 while $k \leq n$

Step 6: Repeat Steps 7 to 1 while $i \leq n$

Step 7: Repeat Steps 8 and 9 while $j \leq n$

Step 8: SET $t[i,j] = t[i][j] \vee (t[i][k] \wedge t[k][j])$

Step 9: INCREMENT j

[END OF LOOP]

Step 10 : INCREMENT i

[END OF LOOP]

Step 11: INCREMENT k

[END OF LOOP]

Step 12: END