

## DATA STRUCTURES SYLLABUS

### Objective:

- ☐ To understand the concepts of basic data structures such as stack, Queues and Linked list.
- ☐ To have general understanding of the network structures through trees and graph.
- ☐ To make the students to understand the basic algorithms for sorting.

**Unit I Basic Concepts:-** Algorithm specification – Data Abstraction – Performance Analysis.

**Arrays and Structures:-** Arrays: Abstract data type – Polynomials – Sparse Matrices – Representation of Multidimensional Arrays. (12L)

**Unit II Stacks and Queues:-** Stacks – Queues – Evaluation of Expressions. **Linked Lists:-** Singly Linked Lists and Chains – Linked Stacks and Queues – Polynomials: Polynomial Representation – Adding Polynomials. Sparse Matrices: Sparse Matrix Representation. – Doubly Linked Lists. (12L)

**Unit III Trees:-** Introduction – Binary Trees – Binary Tree Traversals: Inorder Traversal – Preorder Traversal – Postorder Traversal. Heaps – Binary Search Trees Forests: Transforming a Forest into a Binary Tree. (12L)

**Unit IV Graphs:-** The Graph Abstract Data Type-Elementary Graph Operations – Minimum Cost Spanning Trees: Kruskal's Algorithm – Prim's Algorithm. – Shortest Paths and Transitive Closure: Single Source/ All Destination: Nonnegative Edge Costs - All Pairs Shortest Paths. (12L)

**Unit V Sorting:-** Motivation – Insertion Sort – Quick Sort – Merge Sort: Recursive Merge Sort. – Heap Sort – External Sorting: Introduction – k-way Merging..**Hashing:-** Static Hashing: Hash Tables. (12L) **Text Book:** Fundamentals of Data Structures in C by Ellis Horowitz, Sartaj Sahni,

Susan Anderson- Freed – Second Edition – Universities Press (India) Private Limited. **Reference Books:**

1. Data Structures Using C, Second Edition by Reema Thareja – Oxford University Press
2. Data Structures by Dr N Jeya Prakash – Anuradha Publications

# UNIT-1

## 1. ALGORITHM SPECIFICATION

**Algorithm** is a finite set of instructions to perform a task. All algorithms must satisfy the following criteria.

- **Input:** zero or more quantities externally supplied
- **Output:** Atleast one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** The algorithm terminates after a finite number of steps
- **Effectiveness:** Each operation must be definite and must be feasible.

Eg. void swap(int \*x, int \*y)

```
{  
int temp=*x;  
*x=*y;  
*y=temp;  
}
```

### **Recursive algorithms:**

- A function that is invoked by another function.
- It executes code and then returns control to the calling function.
- A function that calls itself is called direct recursion and that may call other function is called indirect recursion.

### **E.g. Recursive implementation of binary search**

```
int binsearch(int list[])  
{  
int middle;  
If(left<=right)  
{  
middle=(left + right)/2;  
switch(middle) {  
case -1: return binsearch();  
case 0: return middle;  
case 1: return binsearch();  
}}}
```

## 2. Data Abstraction

- The basic data types of c include char, int, float, double, short, long unsigned etc.
- C helps to provide two mechanisms for grouping data together. These are arrays and structures.
- Arrays are collection of elements of the same basic datatype.

Eg.     int list[5];  
       struct{  
              char lastname;  
              int studid;  
              char grade;}student;

- A datatype is a collection of objects and a set of operations that act on those objects.
- An abstract datatype is a datatype that is organized in such a way that the specification of the objects operations on the objects is separated from the representation of operations.
- E.g. Ada has a concept called package and c++ has a concept called class.
- It implies that an Abstract Data Type(ADT) is implementation independent.
- To classify the functions of a datatype into several categories.
  - **creator/constructor**: these functions create a new instance of the designated type.
  - **Transformers**: these functions create an instance of the designated type.
  - **Observers/reporters**: these functions provide information about an instance of the type.

Eg.     ADT natural no  
          Objects: 0 to max  
          Functions:     Boolean IsZero();  
                          Boolean Equal(x, y);  
          End

- There are two main sections in the definition.
- The objects and the functions. Objects are integers. Operations are IsZero() and Equal().

## 3. Performance Analysis

- The criteria on performance evaluation is divided into **two** distinct fields.
- **First** field focuses on obtaining estimates of time and space that are machine independent called as **performance analysis**.

- **Second** field called as **performance measurement** obtains dependent running times. These times used to identify inefficient code segments.
- **Space complexity** of a program is the amount of memory that is needed to run to completion.
- **Time complexity** of a program is the amount of computer time that it needs to run to completion.

1. **Space complexity:** the space needed by a program is the sum of the following components.

**a) Fixed space requirements:** This component refers to space requirements that do not depend on the number and size of the programs input and output.

- These include instruction space, space for simple variables. Fixed size, structured variables and constants.

**b) variable space requirements:** This component consists of the space needed by structured variables whose size depends on the particular instance  $I$  of the problem being solved.

- Additional space required when a function uses recursion.
- If  $n$  is the only instance, the total space requirement  $s(p)$  is

$$S(p)=c+s_p(I) \text{ where } c \text{ is a constant.}$$

Eg.    float abc(float a, float b, float c)  
       {  
       return(a+b+b\*c+(a+b+c))/(a+b)+4.00  
       }

$$S_{abc}(I)=0$$

2. **Time complexity:** The time  $T(p)$  taken by a program is the sum of its compile time and its run time.

- The compile time is similar to the fixed space component.
- When program runs correctly, it is used many times without recompilation.

$$T_p(n)=C_a \text{ ADD}(n)+C_s \text{ SUB}(n)+C_l \text{ LDA}(n)+C_s \text{ STA}(n)$$

where  $C_a, C_s, C_l, C_s$  are constants. ADD, SUB, LDA, STA are the number of additions, subtractions, load and store.



- A program step is syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristic.

Eg. We count a simple assignment statement of the form  $a=2$  as one step and also count a more complex statement as

$A=2*b+3*c/d-e+f/g/a/b/c$  as one step.

Only requirement is that the time required to execute each statement.

Eg.            `float sum(float list[], int n)`  
                  `{`  
                         `for(i=0;i<n;i++)`  
                         `Count t=2;`  
                         `Count t=3;`  
                         `return 0;`  
                  `}`

*The final value will be  $(2n+3)$*

The step count table for matrix addition is

```
void add(int a[][MAX_SIZE]...){
    int i,j;
    for(i=0;i<=rows;i++)
        for(j=0;j<cols;j++)
            c[i][j]=a[i][j]+b[i][j]
}
```

- The best case stepcount is the minimum number of steps that can be executed for the given parameters.
- The worst case stepcount is the maximum number of steps that can be executed for the given parameters.
- The average case stepcount is the average number of steps executed on instances with the given parameters.

## Asymptotic notation( $O, \Omega, \Theta$ )

- To determine step counts is to be able to compare the time complexities of two programs that compute the same function and also to predict the growth in runtime as the instance characteristics change.

- O-notation is one of the very famous mathematical tools available.

$$F(n)=O(g(n))$$

Iff there are two positive constants  $c$  and  $n_0$  so that the following inequality holds for all  $n \geq n_0$

$$F(n) \leq c |g(n)|$$

$F(n)$  is the computing time of some algorithm when the algorithm is run on an input of size ' $n$ '.  $g(n)$  is the standard function like  $n^2$ ,  $n^3$ ,  $n \log n$  etc.

- O-notation has been extremely useful to classify algorithm by their performances.
- This notation helps designers to search for the best algorithms for some problems.
- Complexity expressed in O-notation is only an upper bound and the actual complexity may be much lower.
- This complexity can almost be treated as worst case complexity.
- The constant  $c$  is unknown and is not necessarily small.
- Similarly the constant  $n_0$  is unknown and may not be small.
- **Average case complexity** of the algorithm is much less than its worst case complexity.
- Eg. Quick sort algorithm
- Its worst case complexity is  $O(n^2)$  is  $O(n \log n)$ .
- Its average case, the constants  $c$  and  $n_0$  implicit in the O-notation hide the details of implementation.
- The most common computing times of algorithms are

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

- $\log n$ , if the time complexity of an algorithm is  $\log n$ . The algorithm is said to be logarithmic. The program becomes slightly slower as  $n$  increases
- $n$ , the algorithm is said to be linear
- $n \log n$ , algorithm solves a problem by breaking it up into smaller subproblems.
- $n^2$  the algorithm is said to be quadratic

- $n^3$  the algorithm is said to be cubic as it processes triplets of data items.
- $2^n$ , algorithms with exponential running time.
- Suppose an algorithm consists of three blocks, the first block is for initialization and takes a constant amount of time  $c$ .
- Next block is simple iteration whose time complexity of  $c(n \log n)$ .

#### Fnvals

Logn	N	Nlogn	$n^2$	$n^3$	$2^n$
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	526
4	16	64	256	4096	65536
5	32	160	1024	32768	429496729

## 4. ARRAYS

### ARRAY ABSTRACT DATATYPE

- An array is a set of pairs  $\langle \text{index}, \text{value} \rangle$  such that each index defined has a value associated with it.
- **ADT Array:**
- **Objects:** A set of pairs  $\langle \text{index}, \text{value} \rangle$  for each value of index there is a value from the set item. Index is a finite ordered set of one or more dimensions.
- Eg.  $[0..n-1]$  for one dimensions.  $\{(0,0),(0,1),(0,2),\dots\}$  for two dimensions.
- **Functions** for all  $a \in \text{array}$ ,  $i \in \text{index}$ ,  $x \in \text{item}$ ,  $\text{size} \in \text{integer}$ ,
- **Array Create**( $j$ ,  $\text{list}$ )::= return an array of  $j$  dimensions where  $\text{list}$  is a  $j$  tuple whose  $i$ th element is the size of  $i^{\text{th}}$  dimensions.
- **Item Retrieve**( $a$ ,  $i$ )::=if  $i \in \text{index}$  return the item associated with index value  $i$  in arrays else return error.

**Array store**( $a, i, x$ )::= if  $i \in \text{index}$  return an array that is identical to array except the new pair  $\langle i, x \rangle$  has been inserted else return error.

- The **create**(j, list) function produces a new empty array of the appropriate size.
- **Retrieve** function accepts an array and an index. It returns the value associated with the index, if the index is valid or error if the index is invalid.
- **Store** function accepts an array, an index and an item and returns the original array augmented with the new <index,value> pair.

#### b) Array in C:

- A one dimensional array in C is declared implicitly.

```
int list[5], *ptlist[5];
```

- Eg. There are two arrays declared each containing 5 elements.
- First array defines 5 integers while second array declared 5 pointer integers.
- In c, all arrays start at index 0 so list[0], list[1],....list[n-1]. Similarly ptlist[0:4] contains pointer to an integer.
- When the compiler encounters an array declaration, create list allocates 5 consecutive memory location.
- The address of the first element list[0] is called the base address.
- If the size of an integer is denoted by size of int, then memory address of list[i] is

$\alpha + i * \text{size of (int)}$  where  $\alpha$  is the base address.

## 5. POLYNOMIALS

### POLYNOMIAL ABSTRACT DATATYPE

- One of the simplest and most commonly found data structures is the ordered or linear list.
- Eg. Days of the week={Sunday, Monday.....Saturday}
- We can perform many operations on lists including:
  1. Finding the length n of a list.
  2. Reading the items in a list from left to right.
  3. Retrieving the  $i^{\text{th}}$  item from a list  $0 < i < n$
  4. Replacing the item in the  $i^{\text{th}}$  position of a list  $0 < i < n$
  5. Inserting a new item in the  $i^{\text{th}}$  position of a list
  6. Deleting an item from the  $i^{\text{th}}$  position of a list.

$$\text{Eg. } A(x) = 3x^{20} + 2x^5 + 4$$

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

- The largest exponent of a polynomial is called its degree.
- Coefficients that are zero are not displayed.

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

$$A(x)B(x) = \sum (a_i x^i \cdot \sum b_j x^j)$$

### ADT Polynomial

**Objects:**  $P(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$  a set of ordered pairs of  $\langle e_i, a_i \rangle$  where  $a_i$  is coefficients and  $e_i$  is the exponent of integer  $\geq 0$ .

**Functions:** For all  $\text{poly}, \text{poly1}, \text{poly2} \in \text{Polynomial}$ ,  $\text{coef} \in \text{coefficient}$ ,  $\text{expon} \in \text{exponents}$ .

1. Polynomial  $\text{zero}() ::= \text{return the polynomial } P(x) = 0$
2. Boolean  $\text{Iszero}(\text{poly}) ::= \text{if polynomial returns false else return true.}$
3. Coefficient  $\text{coef}(\text{poly}, \text{expon}) ::= \text{if ( expon } \in \text{ poly) return its coefficient else return zero.}$
4. Exponent  $\text{loadexp}(\text{poly}) ::= \text{return the largest exponent in the polynomial}$
5. Polynomial  $\text{Attach}(\text{poly}, \text{coef}) ::= \text{if (exponent expon)poly return error else return the polynomial poly with the term } \langle \text{coef}, \text{expon} \rangle \text{ inserted.}$
6. Polynomial  $\text{Remove}(\text{poly}, \text{expon}) ::= \text{if(expon } \in \text{ poly) return the polynomial poly with the term whose expon deleted.}$
7. Polynomial  $\text{single mult}(\text{poly}, \text{coef}, \text{expon}) ::= \text{return the polynomial expon, poly, coef, x}$
8. Polynomial  $\text{add}(\text{poly1}, \text{poly2}) ::= \text{return the polynomial poly1+poly2.}$
9. Polynomial  $\text{mult}(\text{poly1}, \text{poly2}) ::= \text{return the polynomial poly1, poly2}$

**End Polynomial.**

### b) Polynomial Representation:

```
typedef struct{
    int degree;
    float coef[MAX_DEG];
}Polynomial
```

### Array Representation of two Polynomials

$$A(x) = 2x^{1000} + 1, B(x) = x^4 + 10x^3 + 3x^2 + 1$$

	Start A	Finish A	Start B			Finish B	Avail
Coef	2	1	1	10	3	1	
Expon	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

### c) Polynomial Addition

Add two polynomials  $D=A+B$

```
void padd(int startA, int FinishA, int StartB, int FinishB, int *StartD, int *FinishD){
```

```
float coef;
```

```
startD=avail;
```

```
while(startA<=FinishA && startB<=FinishB)
```

```
switch(compare(terms[startA].expon, terms[startB].expon)){
```

```
case -1:
```

```
    Attach(terms[startB].coef, terms[startB].expon);
```

```
    StartB++;
```

```
    Break;
```

```
Case 0:
```

```
    Coefficient=terms[StartA].coef+ terms[StartB].coef;
```

```
    If(Coefficient)
```

```
        Attach(coefficient, terms[StartA].expon)
```

```
        StartA++;
```

```
    StartB++;
```

```
    Break;
```

```
Case 1:
```

```
    Attach(terms[StartA].coef,terms[StartA], exponen);
```

```
    StartA++;}
```

```
For(; StartA<=FinishA; StartB++)
```

```
Attach(terms[startB].coef,terms[StartB].expon);
```

```
*FinishD=avail-1;
```

Worst case occurs when

$$A(x) = \sum x^{2i} B(x) = \sum x^{2i+1}$$

The asymptotic computing time of this algorithm is  $O(n+m)$ .

## 6. SPARSE MATRICES:

- A matrix has m rows and n cols of elements. If the first matrix has 5 rows and 3 columns and the second matrix has 6 rows and 6 columns. The total number of elements in such a matrix is m x n. If m=n then the matrix is square matrix.
- When a matrix defined by 2D array as a [MAX\_ROWS][MAX\_COLS] by writing a[i][j] for any element where i is the row index and j is the column index.
- The matrix contains many zero entries So it is a **sparse** matrix. It is represented as <row,col, value> as given here <0,0,5> and <1,2,1>

- $$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- When a sparse matrix is represented as a 2D array we waste space.
- Eg. If only 8 out of 36 possible elements are non zero that is sparse. When dealing with large matrix 1000x1000, if they are sparse, then it is hard to deal.
- A minimal set of operations for representing only non-zero elements are matrix creation, addition, multiplication and transpose.

### a) ADT Sparse Matrix

**Objects:** a set of triples <row, col, val> where row and column form a unique combination and value comes from item.

**Function:** for all a,b ∈ Sparse matrix, x ∈ item, i,j,maxrow,maxcol ∈ index,

**SparseMatrix.create(MaxRow, MaxCol)::=** return a sparse matrix that can hold upto maxitems=maxrowxmaxcol;

**SparseMatrix Transpose(a)::=**return the matrix produced by interchanging row and column value of every tuple.

**SparseMatrix Add(a,b)::=**if the dimensions of a and b are the same return the matrix produced by adding corresponding row and column values ; else return error.

**SparseMatrix(Multiply(a,b))::**if no of columns in a= no of rows in b

Return the matrix d produced by multiplying a and b.

$$D[i][j] = \sum (a[i][k] * b[k][j])$$

Else return error;

**End ADT**

**b) Sparse matrix representation:**

- Use array of triples so as to represent sparse matrix.
- Organise the triple so that the row indices are in ascending order.

```
SparseMatrix create(MaxRow, MaxCol)
```

```
#define maxterms 101
```

```
Typedef struct{
```

```
Int col;
```

```
Int row;
```

```
Int value;}term;
```

```
Term a[Maxterms];
```

```
Maxterms >8
```

Eg.	Row	col	value
A[0]	6	6	8
A[1]	0	0	15
A[2]	0	3	22
A[3]	0	5	-15
A[4]	1	1	11
A[5]	1	2	3
A[6]	2	3	-6
A[7]	4	0	91
A[8]	5	2	28

The triples are ordered by row and within rows by columns.

**c) Transposing a matrix:**

- To transpose a matrix, we interchange the rows and columns.



- Each element  $a[i][j]$  in the original matrix becomes element  $b[i][j]$  in transpose matrix.
- For each row  $i$ 
  - Take element  $\langle i, j, \text{val} \rangle$  and store it as element  $\langle j, i, \text{val} \rangle$  of the transpose.
    - $\langle 0, 0, 15 \rangle$  becomes  $\langle 0, 0, 15 \rangle$
    - $\langle 0, 3, 22 \rangle$  becomes  $\langle 3, 0, 22 \rangle$
    - $\langle 0, 5, -15 \rangle$  becomes  $\langle 5, 0, -15 \rangle$
- To place these triples consecutively in the transpose matrix.
  - For all elements in column  $j$ 
    - Place element  $\langle i, j, \text{val} \rangle$  in element  $\langle j, i, \text{val} \rangle$
- This asymptotic time complexity is  **$O(\text{columns.elements})$** .
- **void transpose(term a[], term b[]){**

```

int n, l, j, current b:
n=a[0].value
b[0]. Col=a[0].row;
b[0].val=n;
if(n>0){
    current=1;
    for(i=0;i<a[0].col;i++)
        for(j=1;j<=n;j++)
            if(a[j].col==l){
                b[current].row=a[j].col;
                b[current].col=a[j].row;
                b[current].value=a[j].value;
                current++;}}}

```
- Fast transpose proceeds by first determining the number of elements in each column of the transpose matrix.
- Asymptotic time complexity is  $O(\text{columns.rows})$ .
- The computing time is  $O(\text{columns+element})$

#### d) Matrix Multiplication:

➤ Given A and B where A is m x n and B is n x p.

➤ The product matrix D has dimension m x p

➤ 
$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

➤ The product of two sparse matrices may no longer be sparse.

➤ 
$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

➤ We compute the elements of D by rows and store them in their proper place.

➤ We pick a row of A and find all the elements in column j

➤ We store the matrices A , B and D in arrays a,b and d.

➤ The overall time of the loops

▪  $O(\sum_{\text{row}}(\text{colsB.termRow}+\text{totalB}))=O(\text{colsB.totalA}+\text{rowsA.totalB})$

➤ Classic multiplication algorithm is

```
For(i=0;i<rowsA;i++)  
    For(j=0;j<colsB;j++){  
        Sum=0;  
        For(k=0;k<colsA;k++)  
            Sum+=(a[i][k]+b[k][j])  
        D[i][j]=sum;}
```

This algorithm takes  $O(\text{rowsA.ColsA.ColsB})$ .

## 7. REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

➤ The array of array representation is to map all elements of a multidimensional array into an ordered or linear list.

➤ Linear list is then stored in consecutive memory as one dimensional array.

➤ If an array is declared  $a[\text{upper}_0][\text{upper}_1] \dots [\text{upper}_{n-1}]$

➤ Number of elements in the array is  $\prod_{i=0}^{n-1} \text{upper}_i$  where  $\prod$  is the product of  $\text{upper}_i$ s

- Eg. A[10][10][10] we require 10.10.10=1000 units of storage to hold the array.
- There are **two** common ways to represent multidimensional arrays. They are row major order and column major order.
- Assume  $\alpha$  is the address of A[0][0] then the address of A[i][0] is  $\alpha + i \cdot \text{upper}_i$  because there are i rows each of size **upper<sub>i</sub>**.
- To represent three dimensional array A[upper<sub>0</sub>][upper<sub>1</sub>][upper<sub>2</sub>], The address of a[i][j][k] is
  - $\alpha + i \cdot \text{upper}_1 \cdot \text{upper}_2 + j \cdot \text{upper}_2 + k$
- The address of a[i<sub>0</sub>][i<sub>1</sub>]..... is  $\alpha + i_0 \cdot \text{upper}_1 \cdot \text{upper}_2 \dots \text{upper}_{n-1} + i_1 \cdot \text{upper}_2 \cdot \text{upper}_3 \dots \text{upper}_{n-1}$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where } a_j = \prod_{i=0}^{n-1} \text{upper}_i \text{ where } 0 \leq j \leq n-1; a_{n-1} = 1$$

\*\*\*\*\*

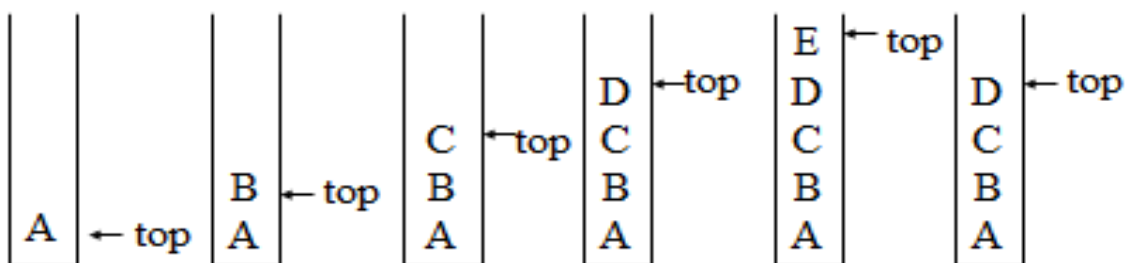
## Unit II

**Stacks and Queues:-** Stacks – Queues – Evaluation of Expressions. **Linked Lists:-** Singly Linked Lists and Chains – Linked Stacks and Queues – Polynomials: Polynomial Representation – Adding Polynomials. Sparse Matrices: Sparse Matrix Representation. – Doubly Linked Lists.

### 1. STACKS

- A stack is an ordered list in which insertions and deletions are made at one end called as the **TOP**.
- Given a stack  $S=\{a_0, a_1, \dots, a_{n-1}\}$  where  $a_0$  is the bottom element,  $a_{n-1}$  is the top element and  $a_i$  is on top of  $a_{i-1}$ .
- Restriction on stack is, if we add the elements A, B, C, D, E to the stack, then E is the first element that can be deleted from the stack.
- Since the last element inserted into a stack is the first element removed, a stack is also known as LIFO (Last In First Out).

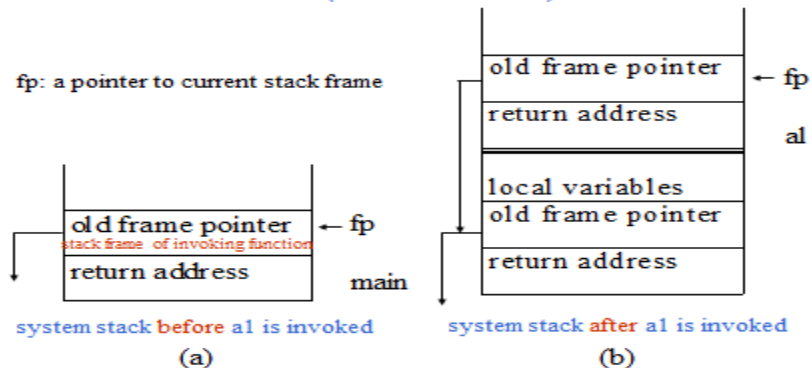
**stack: a Last-In-First-Out (LIFO) list**



Eg. System stack is used by a program at run time to process in calls.

Whenever a function is invoked the program creates a structure referred as activation record or stack frame and places it on top of the system stack.

### an application of stack: stack frame of function call (activation record)



**\*Figure 3.2: System stack after function call a1 (p.103)**

CHAPTER 3

3

- The first or bottom element of the stack is stored in stack[0], second element in stack[1] and  $i^{\text{th}}$  in stack [i-1]. Initially top is set to -1 to denote an empty stack.

### abstract data type for stack

**structure** *Stack* is

**objects:** a finite ordered list with zero or more elements.

**functions:**

for all  $stack \in Stack, item \in element, max\_stack\_size \in \text{positive integer}$

*Stack CreateS(max\_stack\_size) ::=*

create an empty stack whose maximum size is  $max\_stack\_size$

*Boolean IsFull(stack, max\_stack\_size) ::=*

if (number of elements in  $stack == max\_stack\_size$ )

return TRUE

else return FALSE

*Stack Add(stack, item) ::=*

if (IsFull(stack))  $stack\_full$

else insert  $item$  into top of  $stack$  and return

*Boolean IsEmpty(stack) ::=*

if ( $stack == CreateS(max\_stack\_size)$ )

return TRUE

else return FALSE

*Element Delete(stack) ::=*

if (IsEmpty(stack)) return

else remove and return the  $item$  on the top of the stack.

## Add to a stack

```
void add(int *top, element item)
{
    /* add an item to the global stack */
    if (*top >= MAX_STACK_SIZE-1) {
        stack_full( );
        return;
    }
    stack[++*top] = item;
}
```

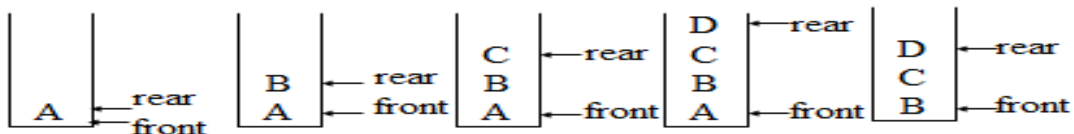
## Delete from a stack

```
element delete(int *top)
{
    /* return the top element from the stack */
    if (*top == -1)
        return stack_empty( ); /* returns and error key */
    return stack[(*top)--];
}
```

## 2. QUEUES

- A queue is an ordered list in which insertions and deletions take place at different ends.
- The end at which new elements are added is called the rear and that from which old elements are deleted is called the front.
- Restrictions on a queue is if we insert A, B, C, D and E then A is the first element deleted from the queue.
- The first element inserted into a queue is the first element removed from queue and is called as First In First Out (FIFO).

### Queue: a First-In-First-Out (FIFO) list



**\*Figure 3.4: Inserting and deleting elements in a queue (p.106)**

## ADT Queue

**structure** *Queue* is

**objects:** a finite ordered list with zero or more elements.

**functions:**

for all *queue*  $\in$  *Queue*, *item*  $\in$  *element*,

*max\_queue\_size*  $\in$  positive integer

*Queue* CreateQ(*max\_queue\_size*) ::=

create an empty queue whose maximum size is

*max\_queue\_size*

*Boolean* IsFullQ(*queue*, *max\_queue\_size*) ::=

if (number of elements in *queue* == *max\_queue\_size*)

**return** *TRUE*

**else return** *FALSE*

*Queue* AddQ(*queue*, *item*) ::=

if (IsFullQ(*queue*)) *queue\_full*

**else** insert *item* at rear of *queue* and return *queue*

*Boolean* IsEmptyQ(*queue*) ::=

if (*queue* == CreateQ(*max\_queue\_size*))

**return** *TRUE*

**else return** *FALSE*

*Element* DeleteQ(*queue*) ::=

if (IsEmptyQ(*queue*)) **return**

**else** remove and return the *item* at front of *queue*.

## ADD FROM A QUEUE

```
void addq(int *rear, element item)
{
/* add an item to the queue */
if (*rear == MAX_QUEUE_SIZE_1) {
    queue_full();
    return;
}
queue[++*rear] = item;
}
```

## DELETE FROM A QUEUE

```
element deleteq(int *front, int rear){
/* remove element at the front of the queue */
if (*front == rear)
    return queue_empty(); /* return an error key */
return queue[++*front];}
```

## CIRCULAR QUEUE:

Front variable points one position counterclockwise from the location of the front element in the queue but rear is unchanged.

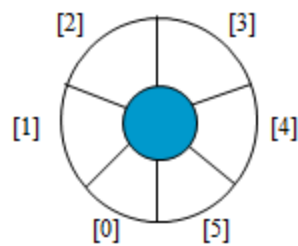
The position next to position MAX\_QUEUE\_SIZE-1 is 0 and the position that precedes 0 is MAX\_QUEUE\_SIZE-1

### Implementation 2: regard an array as a circular queue

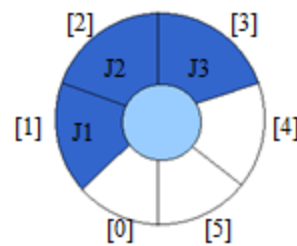
front: one position counterclockwise from the first element

rear: current end

#### EMPTY QUEUE



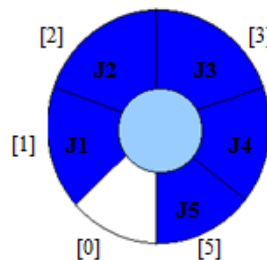
front = 0  
rear = 0



front = 0  
rear = 3

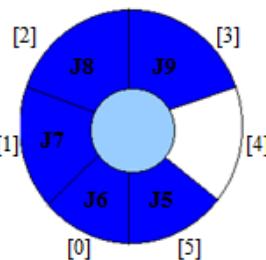
**Problem:** one space is left when queue is full

#### FULL QUEUE



front = 0  
rear = 5

#### FULL QUEUE



front = 4  
rear = 3

Figure 3.7: Full circular queues and then we remove the item (p.110)



### Add to a circular queue

```
void addq(int front, int *rear, element item)
{
    /* add an item to the queue */
    *rear = (*rear + 1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print error */
        return;
}
queue[*rear] = item;
}
```

### Delete from a circular queue

```
element deleteq(int* front, int rear)
{
    element item;
    /* remove front element from the queue and put it in item */
    if (*front == rear)
        return queue_empty( );
    /* queue_empty returns an error key */
    *front = (*front+1) % MAX_QUEUE_SIZE;
    return queue[*front];
}
```

## 3. EVALUATION OF EXPRESSIONS

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

$a = 4, b = c = 2, d = e = 3$

Interpretation 1:

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

Interpretation 2:

$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\ldots$

How to generate the machine instructions  
corresponding to a given expression?

**precedence rule + associative rule**

Token	Operator	Precedence <sup>1</sup>	Associativity
( ) [ ] > .	function call array element struct or union member	17	left-to-right
- ++	increment, decrement <sup>2</sup>	16	left-to-right
- ++ ! - - + & * sizeof	decrement, increment <sup>3</sup> logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >=	relational	10	left-to-right
< <=			
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
⊗	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -=	assignment	2	right-to-left
/= *= %=			
<<= >>=			
&= ^= ⊗			
,	comma	1	left-to-right

1.The precedence column is taken from Harbison and Steele.

2.Postfix form

3.prefix form

**user**

**compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/-ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-


**\*Figure 3.13:** Infix and postfix notation (p.120)

**Postfix:** no parentheses, no precedence

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

\*Figure 3.14: Postfix evaluation (p.120)

## Infix to Postfix Conversion (Intuitive Algorithm)

- Fully parenthesize expression  
 $a / b - c + d * e - a * c \rightarrow$   
 $((((a / b) - c) + (d * e)) - a * c))$
- All operators replace their corresponding right parentheses.  
 $((((a / b) - c) + (d * e)) - a * c))$   

- Delete all parentheses.  
 $ab/c-de*+ac*-$   
two passes

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

\*Figure 3.14: Postfix evaluation (p.120)

The orders of operands in infix and postfix are the same.

$a + b * c, * > +$

Token	Stack [0] [1] [2]	Top	Output
a		-1	a
+	+	0	a
b	+	0	ab
*	+ *	1	ab
c	+ *	1	abc
eos		-1	abc*=

\*Figure 3.15: Translation of  $a+b*c$  to postfix (p.124)

## Rules

- (1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
- (2) ( has low in-stack precedence, and high incoming precedence.

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

$a *_1 (b + c) *_2 d$

Token	Stack [0] [1] [2]	Top	Output
a		-1	a
* <sub>1</sub>	* <sub>1</sub>	0	a
(	* <sub>1</sub> (	1	a
b	* <sub>1</sub> (	1	ab
+	* <sub>1</sub> ( +	2	ab
c	* <sub>1</sub> ( +	2	abc
)	* <sub>1</sub> match )	0	abc+
* <sub>2</sub>	* <sub>2</sub> * <sub>1</sub> = * <sub>2</sub>	0	abc+* <sub>1</sub>
d	* <sub>2</sub>	0	abc+* <sub>1</sub> d
eos	* <sub>2</sub>	0	abc+* <sub>1</sub> d* <sub>2</sub>

CHAPTER 3

\* Figure 3.16: Translation of  $a*(b+c)*d$  to postfix (p.124)

43

Infix	Prefix
$a*b/c$	<u><math>/*abc</math></u>
$a/b-c+d*e-a*c$	<u><math>-+/*abc*de*ac</math></u>
$a*(b+c)/d-g$	<u><math>-/*a+bc dg</math></u>

(1) evaluation

(2) transformation

**\*Figure 3.17: Infix and postfix expressions (p.127)**

## 4. LINKED LISTS

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items.

Each link contains a connection to another link.

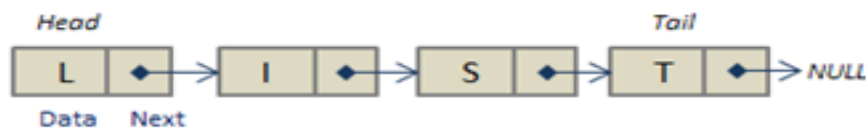
Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List

- Link – Each Link of a linked list can store a data called an element.
- Next – Each Link of a linked list contain a link to next link called Next.
- LinkedList – A LinkedList contains the connection link to the first Link called First.
- Consider the 3 letter English words ending with AT
- (BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT, VAT, WAT)
- To insert GAT it is required to move elements one location higher or lower.
- We must move either HAT, JAT..... or BAT, CAT etc.
- Excessive data movements are required for insertion and deletion.
- An elegant solution to this problem of data movement in sequential representation is achieved using linked representation.
- The elements of the list are stored in a one dimensional array called “Data”. A second array LINK is added to show the array in any order.
- For any i, DATA[i] and LINK[i] comprise a node.

### Non Sequential representation of list representation

DATA	LINK
HAT	15
CAT	4
EAT	9
GAT	1
WAT	0
BAT	3
FAT	6
VAT	7

#### Singly Linked List:



- Nodes do not actually reside in sequential locations.
- Actual locations of nodes are immaterial.

struct Node

```

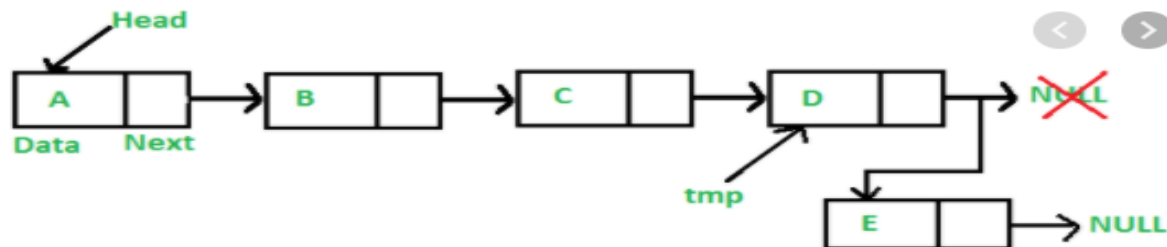
{
    int data;
    struct Node *next;
};
  
```

- In a single linked list each node has exactly one pointer field.
- A chain is a single Linked list that is comprised of zero or more nodes.

**To insert a node into the linked list we need to do following steps.**

For example the data item D has to be inserted between C and E

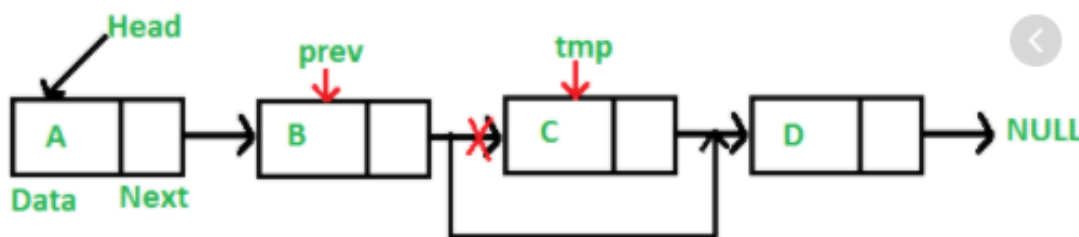
- (i) get a node which is currently unused; let its address be X;
- (ii) set the DATA field of this node to D;
- (iii) set the LINK field of X to point to the node after C which contains E;
- (iv) set the LINK field of the node containing C to X.



To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.

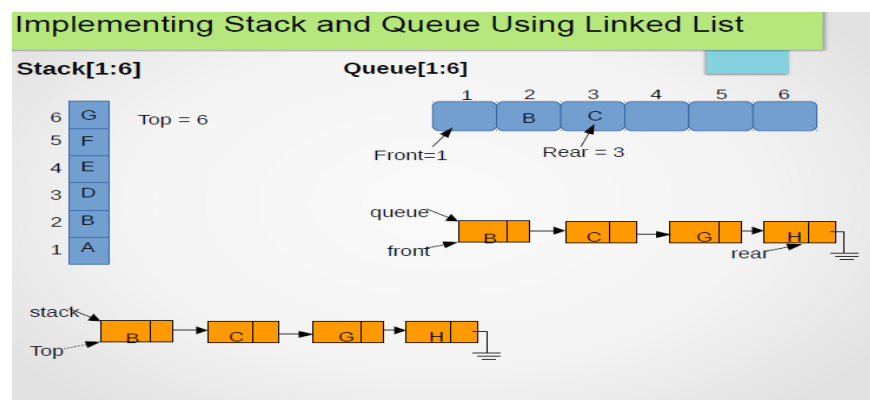
To delete an element from a single linked list. To delete C from the list. Find the element that immediately precedes C which is B and set link to the position of D which is after C.



## 5. LINKED STACKS AND QUEUES:

A **linked stack** is a linear list of elements commonly implemented as a singly linked list whose start pointer performs the role of the top pointer of a stack

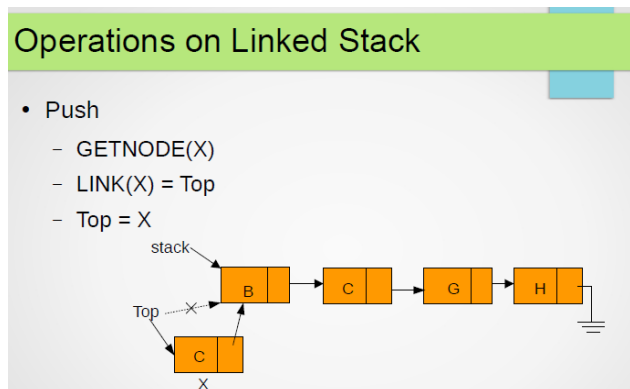
• A **linked queue** is also a linear list of elements commonly implemented as a singly linked list but with two pointers viz., FRONT and REAR. The start pointer of the singly linked list plays the role of FRONT while the pointer to the last node is set to play the role of REAR.



When several stacks and queues coexisted, there was no efficient way to represent them sequentially.

We can easily add or delete a node from the top of the stack.

We can easily add or delete a node to the rear of the queue and add or delete a node at the front.



**Algorithm: Push item ITEM into a linked stack S with top pointer TOP**

**procedure** PUSH\_LINKSTACK (TOP, ITEM)

*/\* Insert ITEM into stack \*/*

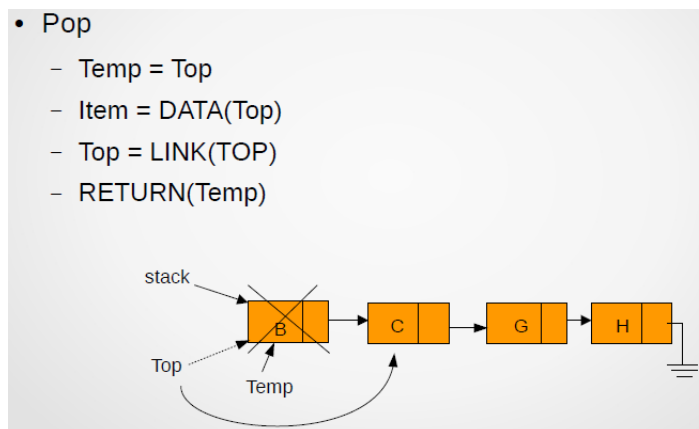
**Call** GETNODE(X)

DATA(X) = ITEM */\*frame node for ITEM \*/*

LINK(X) = TOP */\* insert node X into stack \*/*

TOP = X */\* reset TOP pointer \*/*

**end** PUSH\_LINKSTACK.



**Algorithm: Pop from a linked stack S and output the element through ITEM**

**procedure** POP\_LINKSTACK(TOP, ITEM)

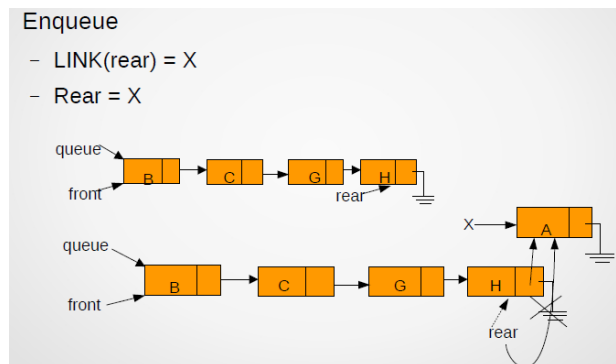
*/\* pop element from stack and set ITEM to the element \*/*



```

if (TOP = 0) then call LINKSTACK_EMPTY
/* check if linked stack is empty */
else { TEMP = TOP
ITEM = DATA(TOP)
TOP = LINK(TOP)
}
call RETURN(TEMP) ;
end POP_LINKSTACK.

```



**Algorithm: Enqueue an ITEM into a linked list queue Q**

**procedure** INSERT\_LINKQUEUE(FRONT, REAR, ITEM)

**Call** GETNODE(X);

DATA(X) = ITEM;

LINK(X) = NIL; /\* Node with ITEM is ready to be inserted into Q \*/

**if** (Queue = 0) **then**

• Queue = FRONT = REAR = X;

/\* If Q is empty then ITEM is the first element in the queue Q

**else** { LINK(REAR) = X;

REAR = X

}

**end** INSERT\_LINKQUEUE.

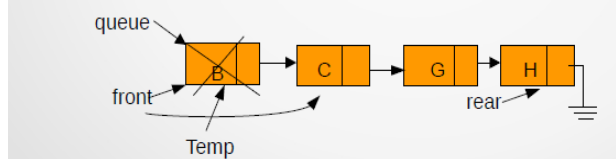
• **Dequeue**

- Temp = front

- front = Link(front)

- Item = DATA(Temp)

- RETURN(Temp)



### Algorithm: Dequeue an element from the linked queue Q

**procedure** DELETE\_LINKQUEUE (FRONT,ITEM)

**if** (FRONT = 0) **then call** LINKQUEUE\_EMPTY;

/\* Test condition to avoid deletion in an empty

queue \*/

**else** { TEMP = FRONT;

ITEM = DATA (TEMP);

FRONT = LINK (TEMP);

}

**call RETURN** (TEMP); /\* return the node TEMP to  
the free pool \*/

**end** DELETE\_LINKQUEUE

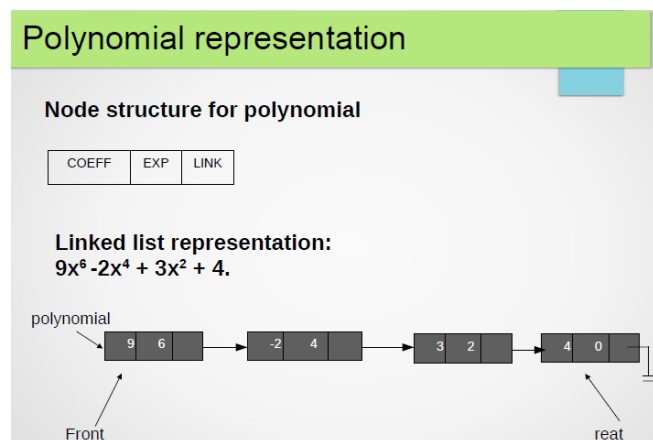
## 6. POLYNOMIAL ADDITION

To represent any number of different polynomials as long as their combined size does not exceed our block of memory. In general, we want to represent the polynomial

$$A(x) = a^m x e^m + \dots + a^1 x e^1$$

where the  $a^i$  are non-zero coefficients with exponents  $e^i$  such that  $e^m > e^{m-1} > \dots > e^2 > e^1 \geq 0$ . Each term will be represented by a node. A node will be of fixed size having 3 fields which represent the coefficient and exponent of a term plus a pointer to the next term

COEF	EXP	LINK



For instance, the polynomial  $A = 3x^{14} + 2x^8 + 1$  would be stored as while  $B = 8x^{14} - 3x^{10} + 10x^6$ . To add two polynomials together examine their terms starting at the nodes pointed to by  $A$  and  $B$ . Two pointers  $p$  and  $q$  are used to move along the terms of  $A$  and  $B$ . If the exponents of two terms are equal, then the coefficients are added and a new term created for the result. If the exponent of the current term in  $A$  is less than the exponent of the current term of  $B$ , then a duplicate of the term of  $B$  is created and attached to  $C$ . The pointer  $q$  is advanced to the next term. Similar action is taken on  $A$  if  $\text{EXP}(p) > \text{EXP}(q)$

**Input:**

1st number =  $5x^2 + 4x^1 + 2x^0$

2nd number =  $5x^1 + 5x^0$

Output:

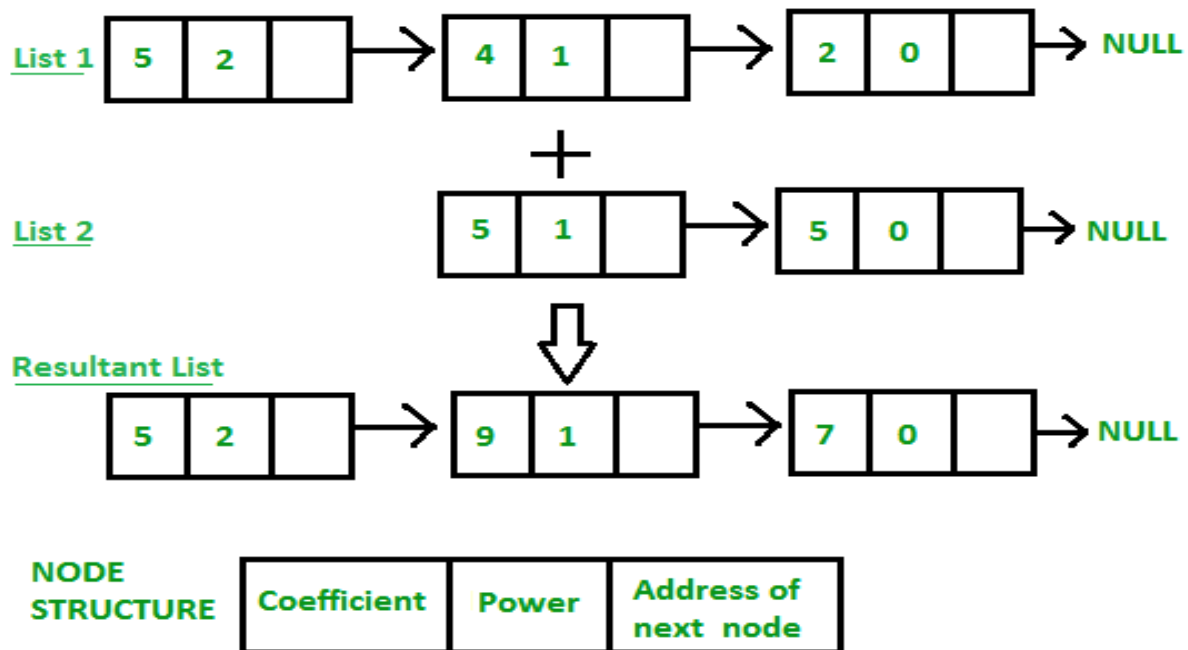
$5x^2 + 9x^1 + 7x^0$

Input:

1st number =  $5x^3 + 4x^2 + 2x^0$

2nd number =  $5x^1 + 5x^0$

Output:



```
Void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
{
    // If power of 1st polynomial is greater than 2nd, then store 1st as
    // and move its pointer
    if(poly1->pow > poly2->pow)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }

    // If power of 2nd polynomial is greater than 1st, then store 2nd as
    // and move its pointer
    else if(poly1->pow < poly2->pow)
    {
```

```

        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }

    // If power of both polynomial numbers is same then add their Coefficients
    else
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff+poly2->coeff;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }

    // Dynamically create new node
    poly->next = (struct Node *)malloc(sizeof(struct Node));
    poly = poly->next;
    poly->next = NULL;
}
while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }
    if(poly2->next)
    {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }
    poly->next = (struct Node *)malloc(sizeof(struct Node));
    poly = poly->next;
    poly->next = NULL;
}
}

// Display Linked list
void show(struct Node *node)
{
    while(node->next != NULL)
    {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if(node->next != NULL)
            printf(" + ");
    }
}

// Driver program
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

```

```

// Create first list of 5x^2 + 4x^1 + 2x^0
create_node(5,2,&poly1);
create_node(4,1,&poly1);
create_node(2,0,&poly1);

// Create second list of 5x^1 + 5x^0
create_node(5,1,&poly2);
create_node(5,0,&poly2);

printf("1st Number: ");
show(poly1);

printf("\n2nd Number: ");
show(poly2);

poly = (struct Node *)malloc(sizeof(struct Node));

// Function add two polynomial numbers
polyadd(poly1, poly2, poly);

// Display resultant List
printf("\nAdded polynomial: ");
show(poly);

return 0;
}

```

Output:

```

1st Number: 5x^2 + 4x^1 + 2x^0
2nd Number: 5x^1 + 5x^0
Added polynomial: 5x^2 + 9x^1 + 7x^0

```

## 7. SPARSE MATRICES

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as **sparse matrix**.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements.

In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate  $100 \times 100 \times 2 = 20000$  bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

## Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation
2. Linked Representation

### Method 1 : Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0<sup>th</sup> row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

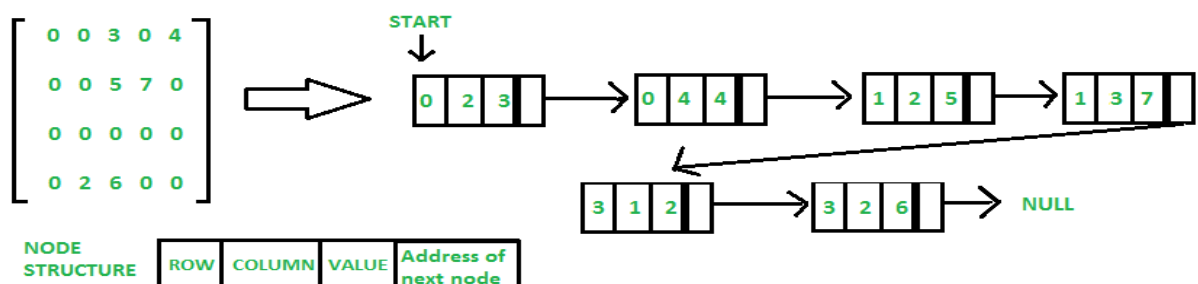
In above example matrix, there are only 6 non-zero elements ( those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image.

Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

### Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)
- Next node: Address of the next node



### Why to use Sparse Matrix instead of simple matrix ?

- Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those non-zero elements.

- Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

### Operations of Sparse Matrix

- Add
- Transpose and
- Multiply

Given two sparse matrices, perform the operations such as add, multiply or transpose of the matrices in their sparse form itself.

The result should consist of three sparse matrices, one obtained by adding the two input matrices, one by multiplying the two matrices and one obtained by transpose of the first matrix.

Example: Note that other entries of matrices will be zero as matrices are sparse.

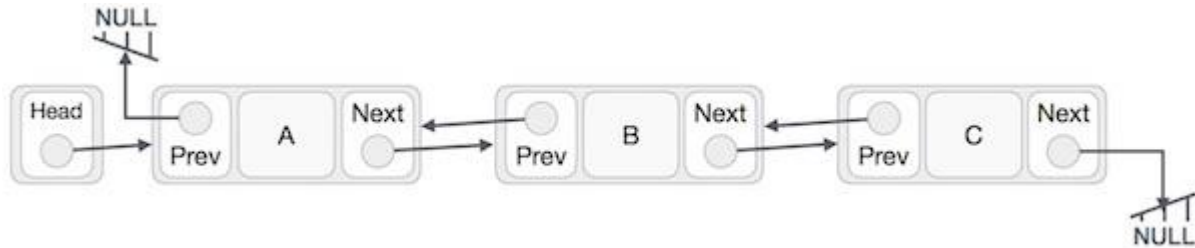
Operations on Sparse Matrices								
Row	Column	Value	Row	Column	Value	Row	Column	Value
1	2	10	1	3	8	1	2	10
1	4	12	2	4	23	1	3	8
3	3	5	3	3	9	1	4	12
4	1	15	4	1	20	2	4	23
4	2	12	4	2	25	3	3	14
Input : Matrix 1: (4x4)			Matrix 2: (4x4)			4	1	35
						4	2	37
			Row	Column	Value	Result of Transpose		
Result of Multiplication			1	1	240			
			1	2	300			
			1	4	230			
			3	3	45			
			4	3	120			
			4	4	276			
			Row	Column	Value	Result of Addition		
			1	4	15			
			2	1	10			
			2	4	12			
			3	3	5			
			4	1	12			

## 8. DOUBLY LINKED LIST

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- Link – Each link of a linked list can store a data called an element.
- Next – Each link of a linked list contains a link to the next link called Next.
- Prev – Each link of a linked list contains a link to the previous link called Prev.
- LinkedList – A Linked List contains the connection link to the first link called First and to the last link called Last.
-

## Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

## Basic Operations

Following are the basic operations supported by a list.

- Insertion – Adds an element at the beginning of the list.
- Deletion – Deletes an element at the beginning of the list.
- Insert Last – Adds an element at the end of the list.
- Delete Last – Deletes an element from the end of the list.
- Insert After – Adds an element after an item of the list.
- Delete – Deletes an element from the list using the key.
- Display forward – Displays the complete list in a forward manner.
- Display backward – Displays the complete list in a backward manner.

## Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

```
//insert link at the first location
void insertFirst(int key, int data) {
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;
    if(isEmpty()) {
        //make it the last link
        last = link;
    }
}
```



```

} else {
    //update first prev link
    head->prev = link;
}
//point it to old first link
link->next = head;
    //point first to new first link
head = link;
}

```

## Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

```

//delete first item
struct node* deleteFirst() {
    //save reference to first link
    struct node *tempLink = head;
        //if only one link
    if(head->next == NULL) {
        last = NULL;
    } else {
        head->next->prev = NULL;
    }
    head = head->next;
        //return the deleted link
    return tempLink;
}

```

## Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

```

//insert link at the last location
void insertLast(int key, int data) {
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //make link a new last link
        last->next = link;
        //mark old last node as prev of new link
        link->prev = last;
    }
    //point last to new last node
    last = link;}

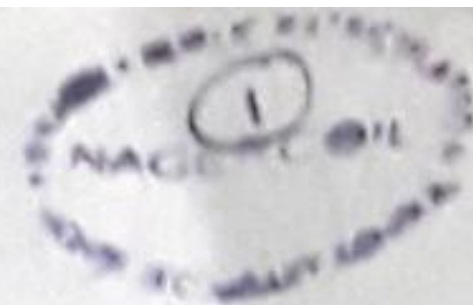
```



# Data Structures

## UNIT-III

### TREES

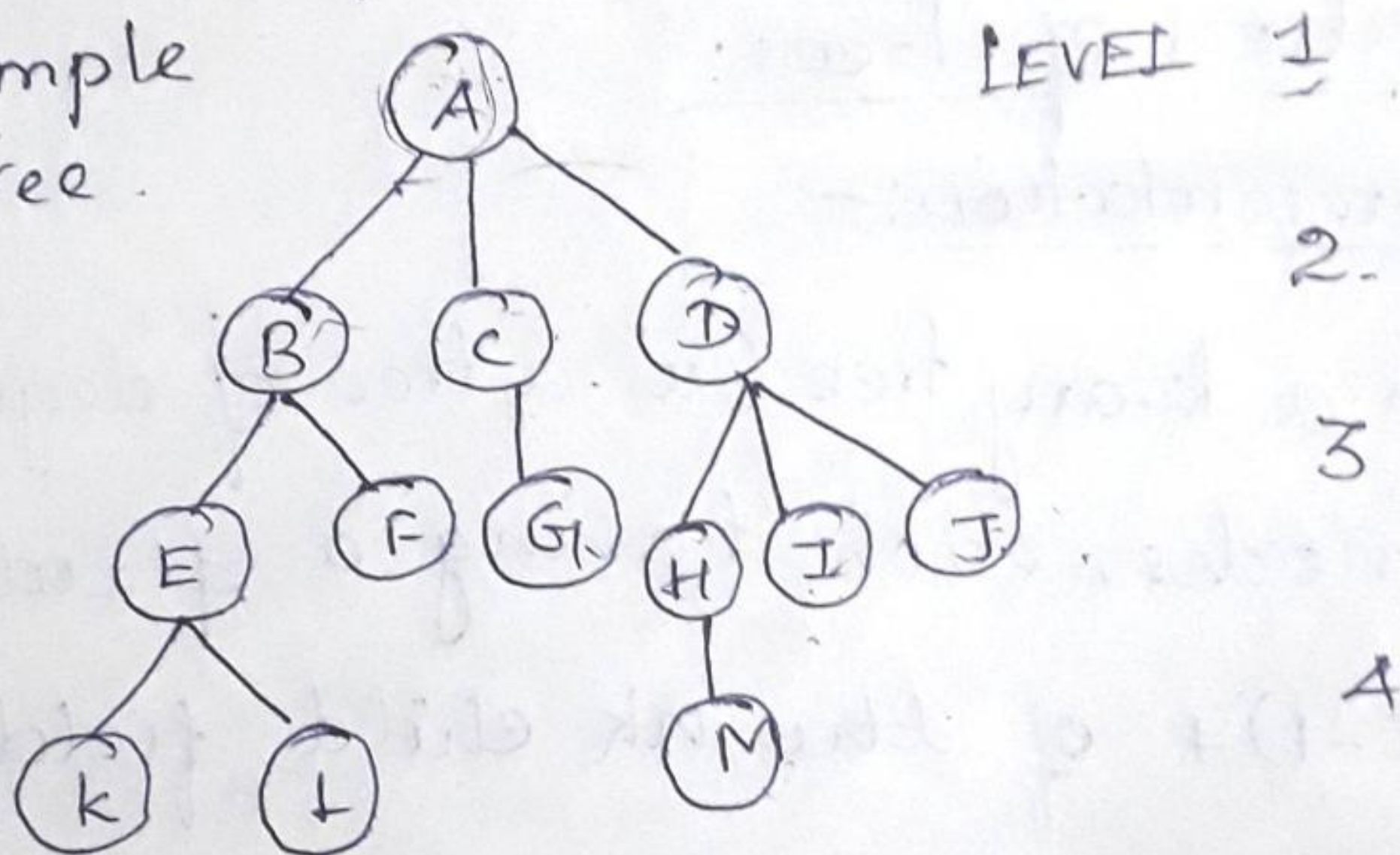


#### 1. INTRODUCTION: TREES

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

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

Fig: Sample Tree.



- The number of subtree of a node called degree. eg: degree of A is 3 & C is 1 & F is zero.
- Nodes that have degree zero called leaf or terminal nodes. other nodes are nonterminal.
- The roots of the subtrees of a node X are the children of X. eg: children of D are H, I and J.



- The degree of a tree is the maximum of the degree of the nodes in the tree  
eg: degree of sample tree is 3.
- The ancestors of the node are all the nodes along the path from root to that node. eg: ancestors of M are A, D + H from sample tree
- level of a node is defined by letting the root be at level one and the children are at level  $l+1$ .

## Representation of Trees:

### (1) List Representation:-

If  $T$  is a  $k$ -ary tree (ie. a tree of degree  $k$ ) with  $n$  nodes, each having a fixed size then  $n(k-1) + 1$  of the  $nk$  child fields are 0,  $n \geq 1$

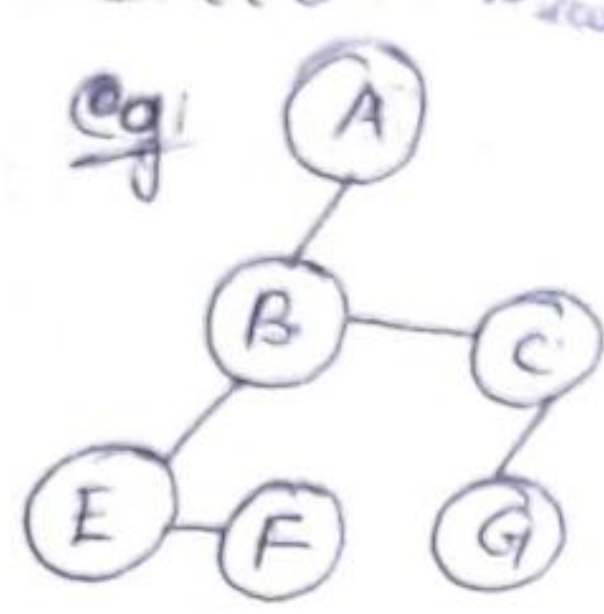
Data	child 1	child 2	...	child $k$
------	---------	---------	-----	-----------

Each child field is used to point to a subtree.



## (2) Left child Right sibling Representation -

Data	
Left child	Right sibling



The left child field of each node points to its leftmost child and the right sibling field point to its closest right sibling.

## (3) Representation as a Degree Two tree

In the degree two representation, two children of a node, as left and right children. The right sibling pointers in a left child-right sibling tree clockwise by 45 degrees. Left child right child tree also known as binary trees.

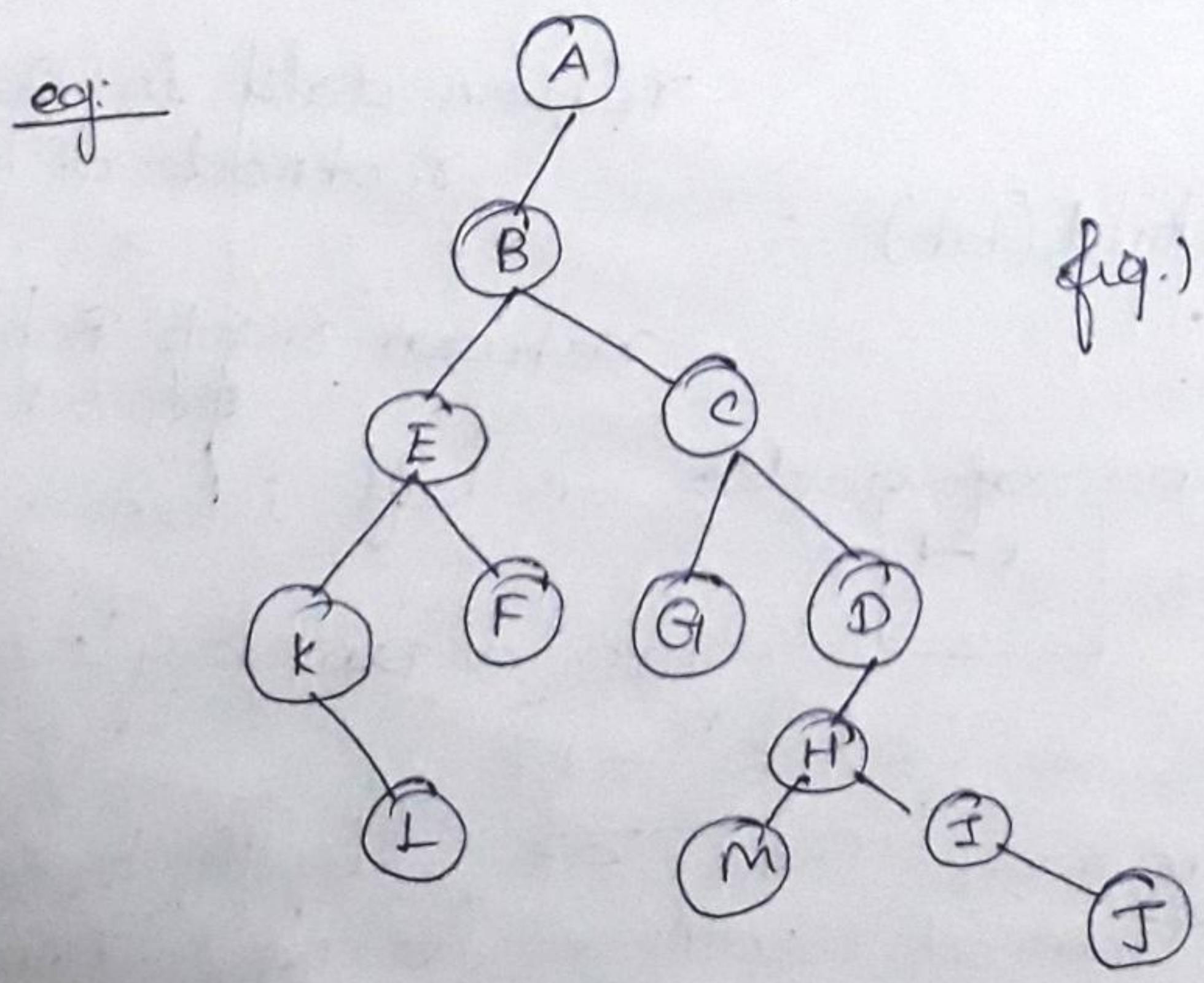


fig.) Left child Right child tree representation



a) Binary Tree ADT

a) Binary Tree ADT

ADT Binary-Tree ~~is~~ 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  $bt, bt_1, bt_2 \in \text{BinTree}$   
item  $\in \text{element}$

BinTree Create() :: Creates an empty binary tree

```

Boolean IsEmpty(bt) ::= if (bt == empty bt)
                        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 root node contains data item.

BinTree Lchild(bt) :- If (IsEmpty(bt)) return error  
else return left subtree of bt.

element Data (bt) ::= If .  
return data in the  
rootnode of bt.

Bin Tree Rchild(bt) ::= " return right subtree of bt.

### b) Properties

1. The max. no. of nodes on left i on a subtree is  $2^i - 1$

Subtree is  $2^k$  in a binary tree  
2. " " " $2^k - 1$ " of depth  $k$  is

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



### BINARY TREE TRAVERSALS 3.

\* Visiting each node in a tree exactly once is known as traversing a tree. A full traversal produces a linear order for nodes in a tree.

Three different types of traversals were performed.

- (i) Inorder traversal
- (ii) postorder traversal
- (iii) preorder traversal

#### a) Inorder traversal:-

Inorder traversal calls for moving down the tree towards the left. ~~until you~~ Then you "visit" the node, move one node to right and continue.

If you cannot move to the right go back one more node.



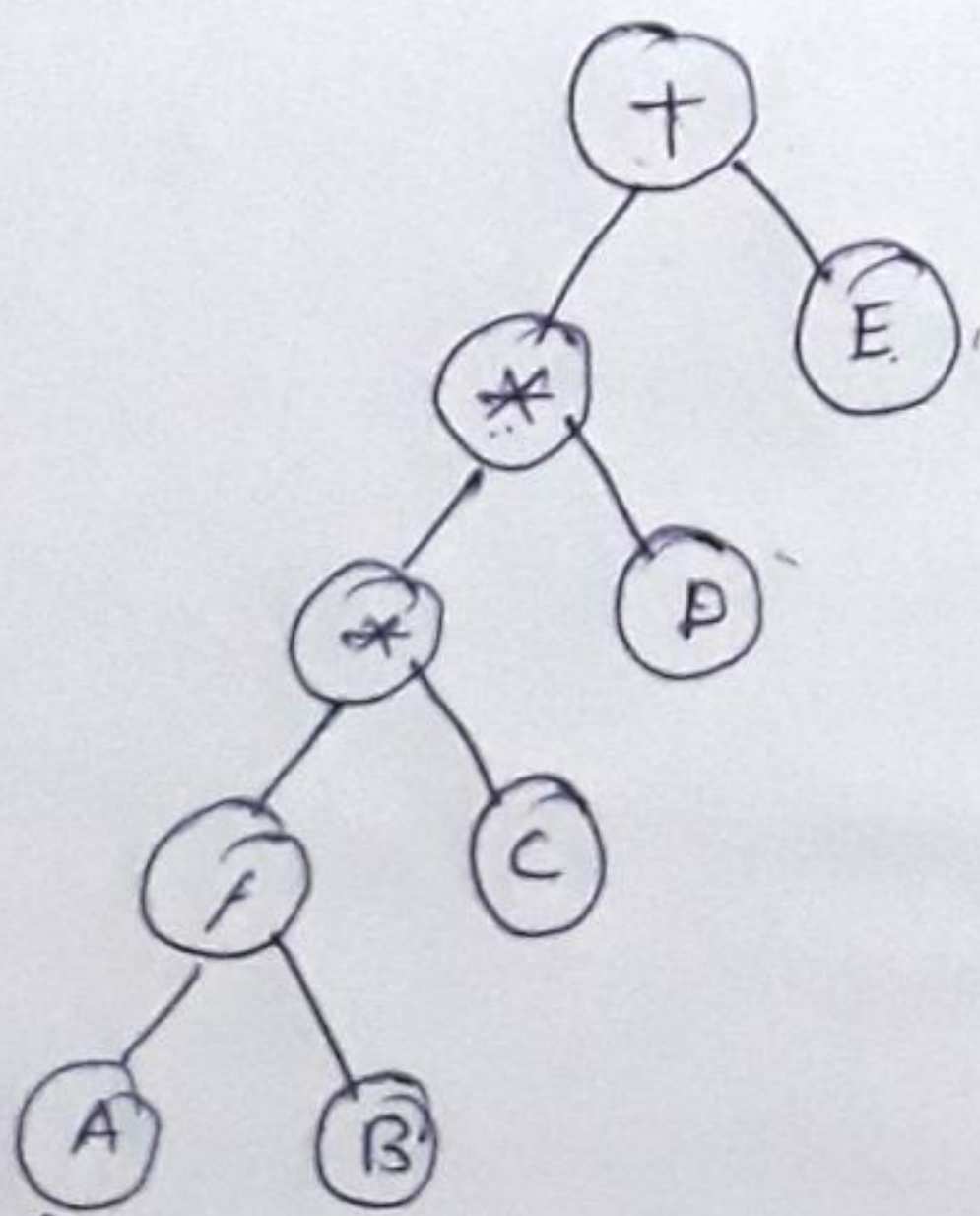
preorder traversal: b) Pre-order Traversal



In preorder traversal firstly "Visit a node, traverse left and then move to the right node and begin again".

```
void preorder (tree pointer ptr)
{
    if (ptr)
    {
        printf ("%d", ptr->data);
        preorder (ptr->leftchild);
        preorder (ptr->rightchild);
    }
}
```

eg:



In preorder traversal output in order

+ \* \* / A B C D E

It is an prefix form of expression

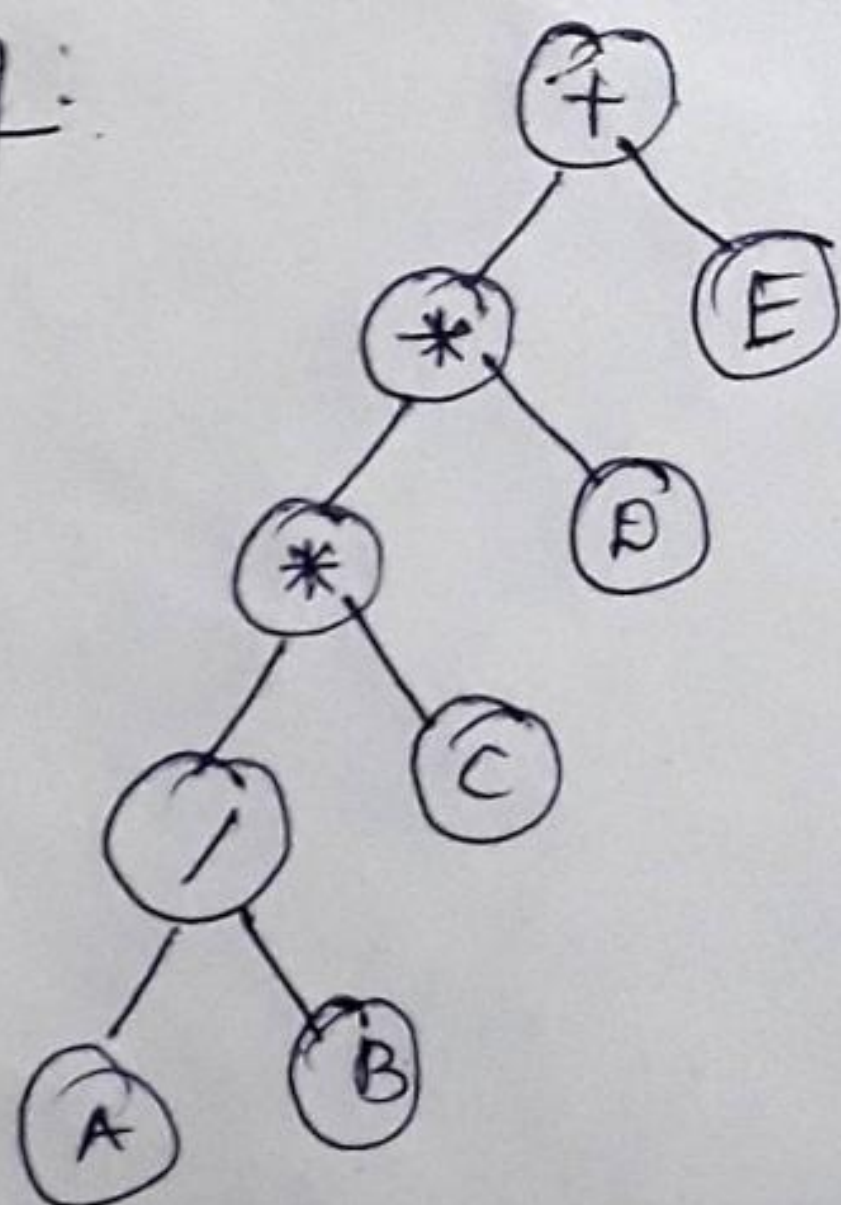


## postorder traversal:

In postorder traversal firstly visit the left node, then traverse to the right node and then move to the root node.

```
void postorder (tree pointer ptr)
{
    if (ptr)
    {
        postorder (ptr → left child);
        postorder (ptr → Right child);
        printf ("%d", ptr → data);
    }
}
```

eg:



The postorder traversal output is

**AB / C \* D \* E +**

It is a postfix form of expression.



## 4. HEAPS      4. HEAPS

### a) priority queue:

→ Heaps are frequently used to implement priority queue.

→ In this kind of queue, the element to be deleted is the one with highest (or lowest) priority.

```
void insertRight (threaded pointer s,
                  threaded pointer r)
{
    threaded pointer = temp;
    r → right child = parent → right child;
    r → rightThread = parent → rightThread;
    r → left child = parent;
    r → leftThread = TRUE;
    s → right child = child;
    s → rightThread = FALSE;
    if (! r → rightThread)
    {
        temp = insucc(r);
        temp → left child = r;
    }
}
```

program: Right insertion in a threaded binary tree.



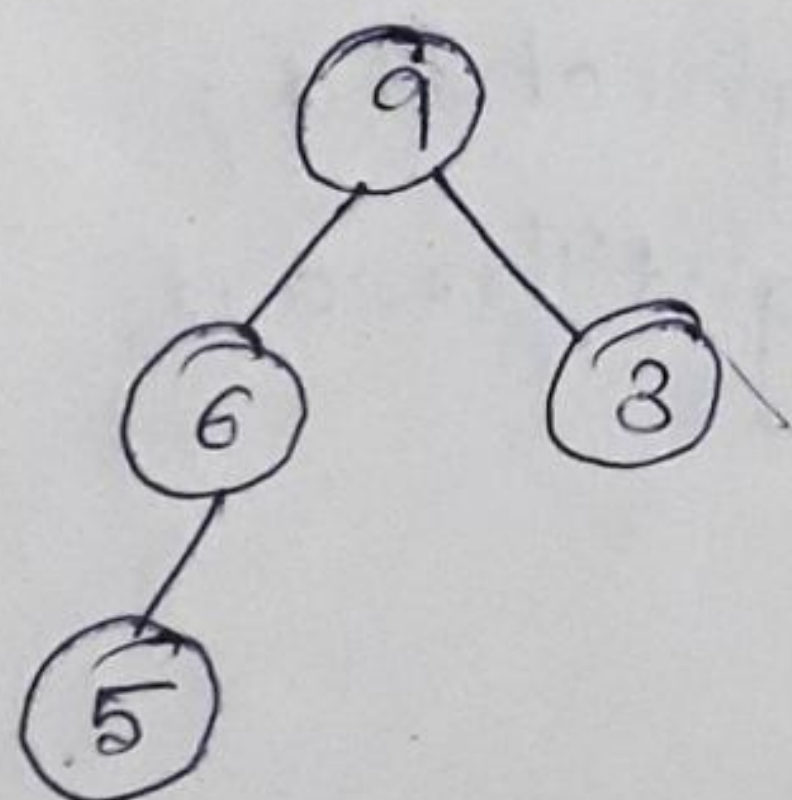
## Definition of a Max Heap:-

A max(min) tree is a tree in which the key value in each node is no smaller (larger) than the key value in its children.

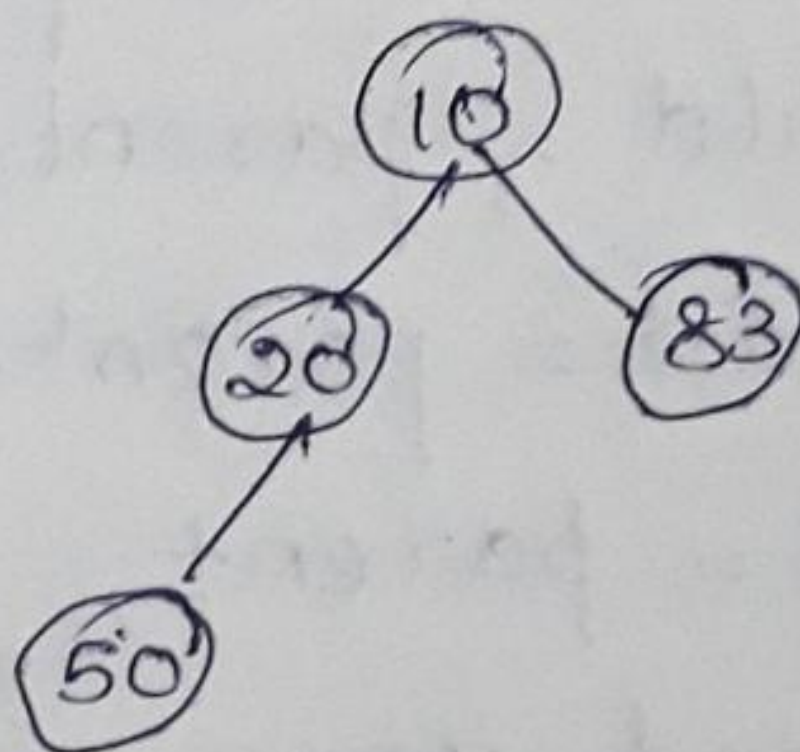
→ A max heap is a complete binary tree that is also a max tree.

→ A min heap is a complete binary tree that is also a min tree.

max heap:



min heap:



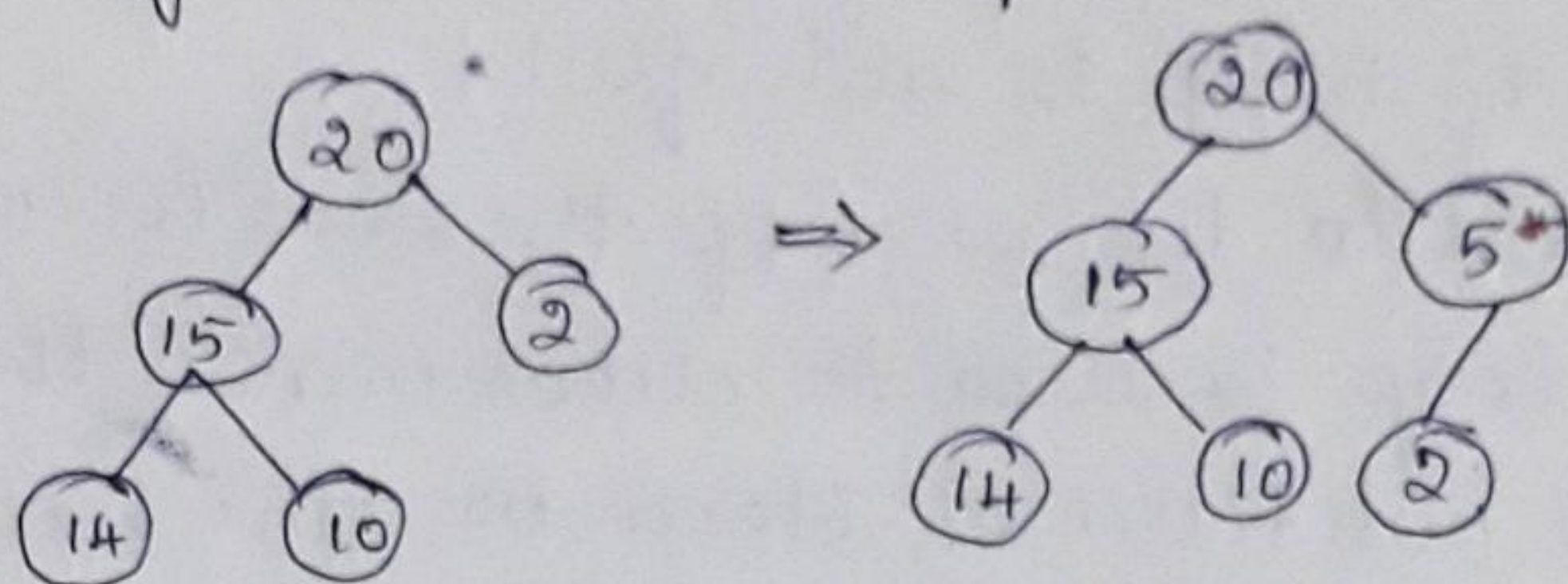
root of a max tree is largest and the root of the min tree is smallest.

## Insertion into a max heap:-

→ To determine the correct place for the element that is being inserted, we use a bubbling up process that begins at the new node of the tree and move towards the root.



Eg: Inserting the new element '5' in ~~binary tree~~ max heap.



- The new element 5 cannot be inserted as the left child of 2.
- So, 2 is moved down to its left child and placing 5 at the older position of 2 in max heap.

```
void push(element item, int *n)
```

```
{
    int i;
```

```
    if (HEAP-FULL(*n))
```

```
    {
```

```
        printf(stderr, "The heap is full. \n");
```

```
        exit(EXIT-FAILURE);
```

```
    }
```

```
    i = ++(*n);
```

```
    while ((i != 1) && (item.key > heap(i/2).key))
```

```
    {
```

```
        heap(i) = heap(i/2);
```

```
        i /= 2;
```

```
    }
```

```
    heap[i] = item;
```

```
}
```



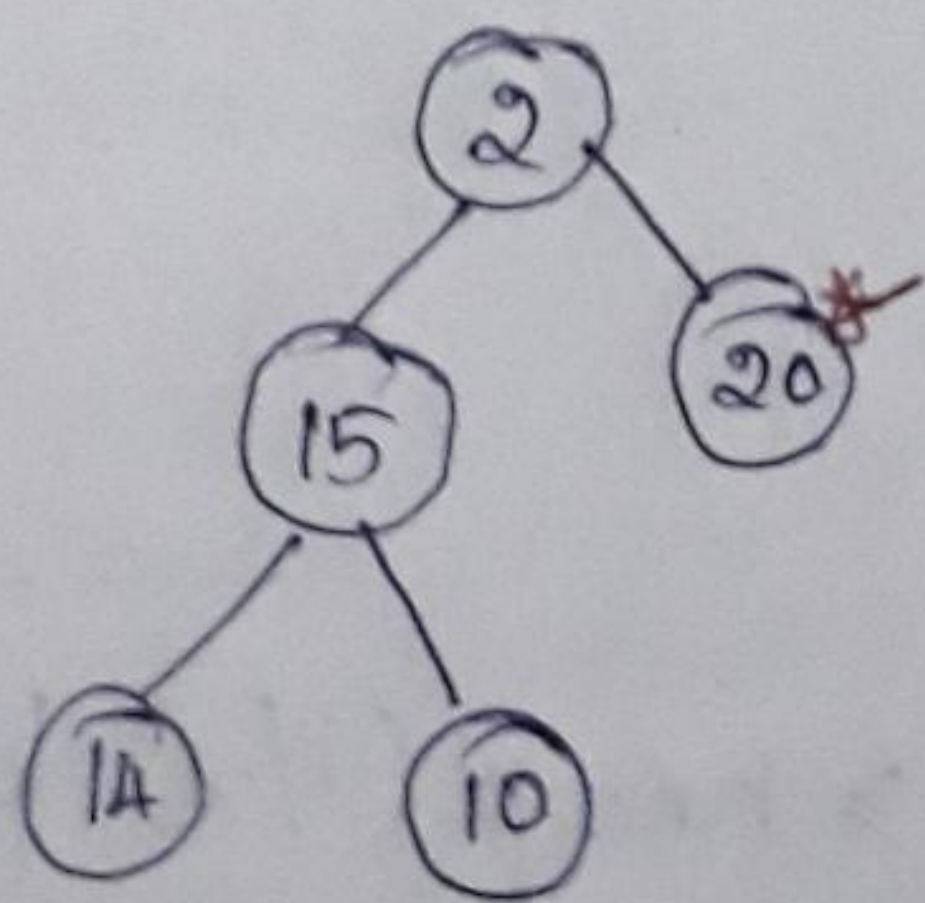
### Analysis of push:

- The function push first checks for a full heap. If heap is not full.
- we set  $i$  to the size of the new heap ( $n+1$ ).
- while loop is used to determine the correct position of item in the heap.
- ~~while loop is iterated  $O(\log_2 n)$  times.~~
- Heap is a complete binary tree with  $n$  elements with height of  $\boxed{\log_2(n+1)}$
- while loop is iterated to  $\boxed{O(\log_2 n) \text{ times}}$

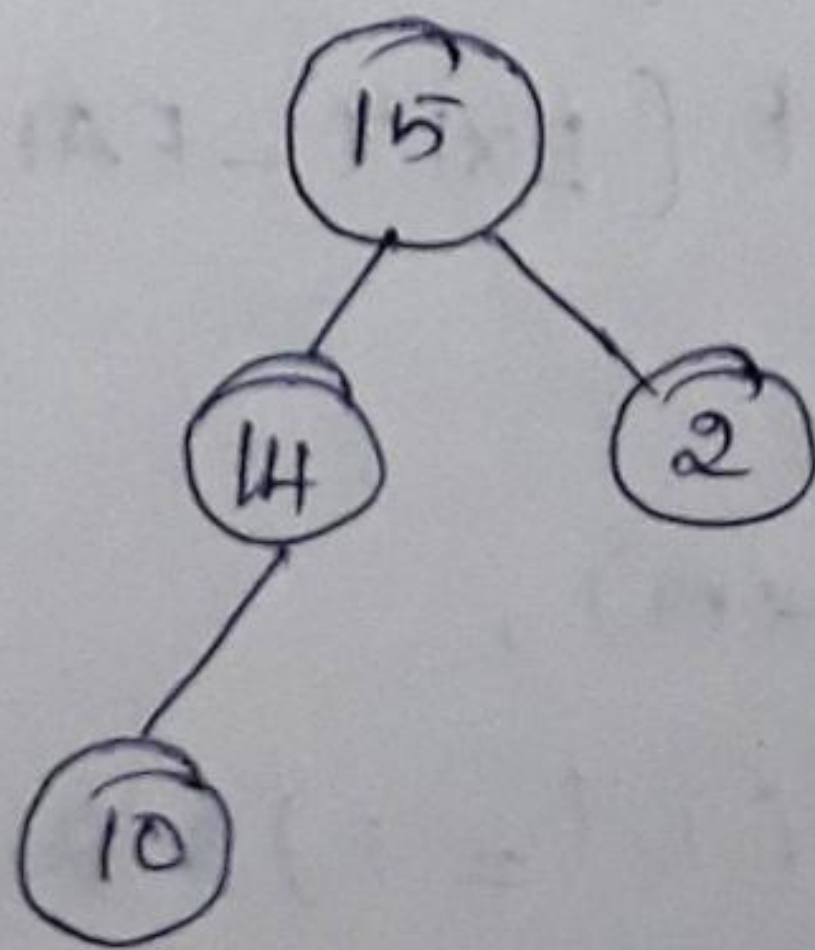
### Deletion from a max heap:-

- when an element is to be deleted from a max heap, it is taken from the root of the element.

eg: deleting the element '20' from the ~~max~~ heap



⇒



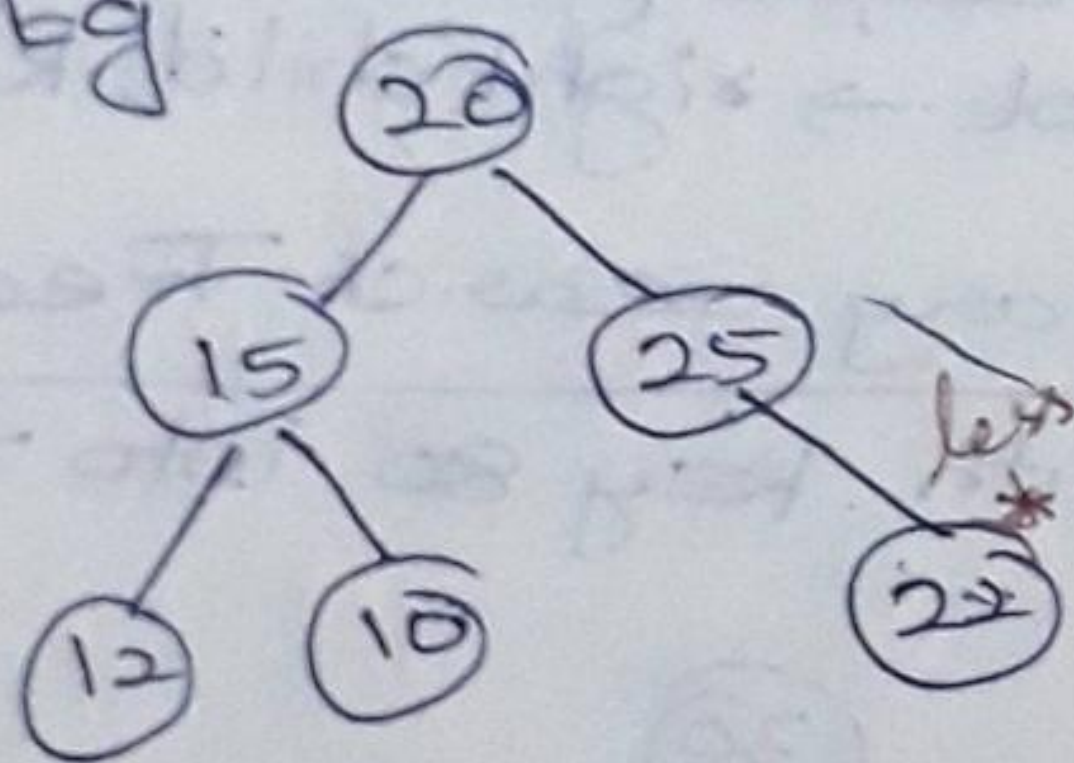


## 5 BINARY SEARCH TREES

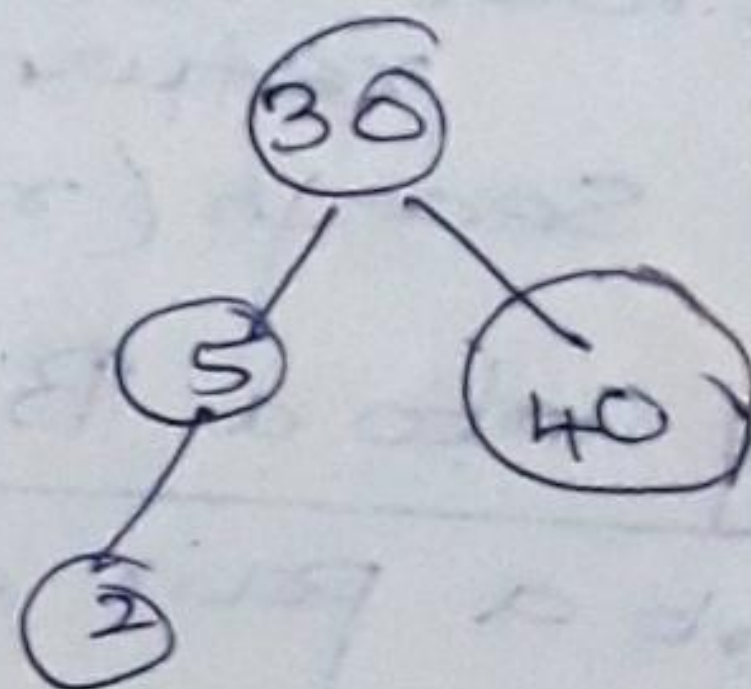
Defn A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

- ① Each node has exactly one key and the keys in the tree are distinct.
- ② The keys in the left subtree are smaller than the key in the root.
- ③ The keys in the right subtree are larger than the key in the root.
- ④ The left and right subtrees are also binary search trees.

Eg.

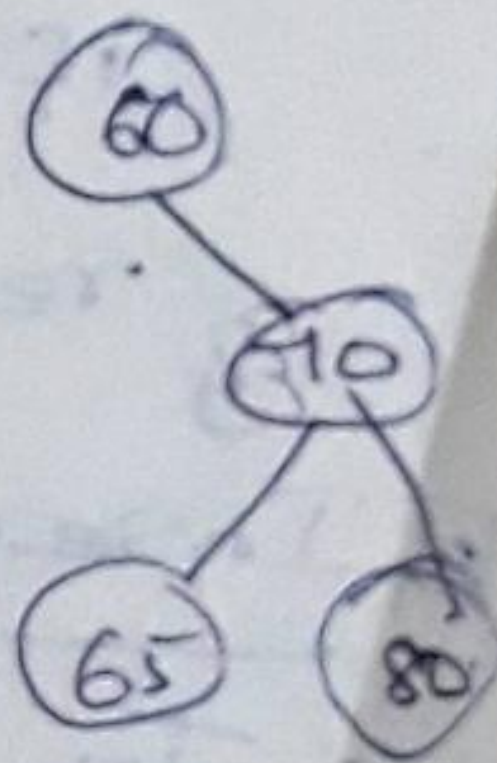


Not a binary search tree as right subtree has key value 22.



(b)

(b) & (c) are binary search trees.



(c)

### ① Searching a Binary Search tree:-

- \* Binary search tree is recursive..
- \* Search for a node whose key is (k).
- \* Start from the root of the binary search tree.

\* If root is NULL, search tree contains no nodes, search is unsuccessful.



\* Otherwise compare  $k$  with the key in root:

① If  $k$  equals the root's key, then search terminates successfully.

② If  $k$  is less than root's key, then no element in the right subtree can have a key value equal to  $k$ .

\* Search the left subtree of the root

③ If  $k$  is larger than root's key value, search the right subtree of the root.

\* Function search recursively searches the subtrees.

element \*search (tree pointer root, int key)

{ if (!root) return NULL;

if ( $k == \text{root} \rightarrow \text{data} \cdot \text{key}$ ) return  $k(\text{root} \rightarrow \text{data})$ ;

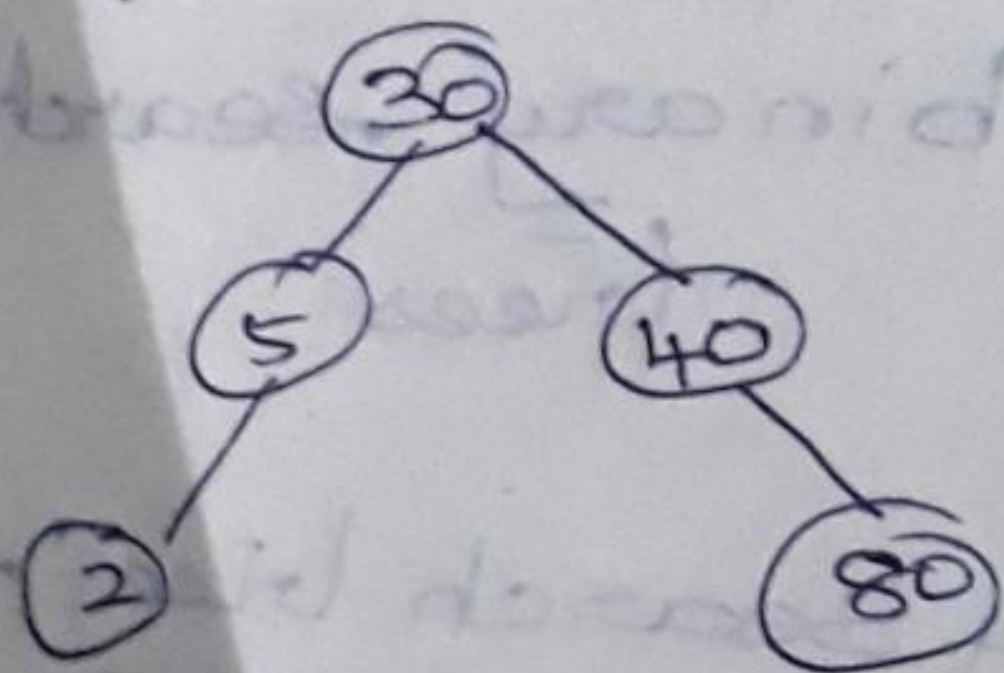
if ( $k < \text{root} \rightarrow \text{data} \cdot \text{key}$ ) return search (root  $\rightarrow$  left child,  $k$ );

return search (root  $\rightarrow$  right child,  $k$ );

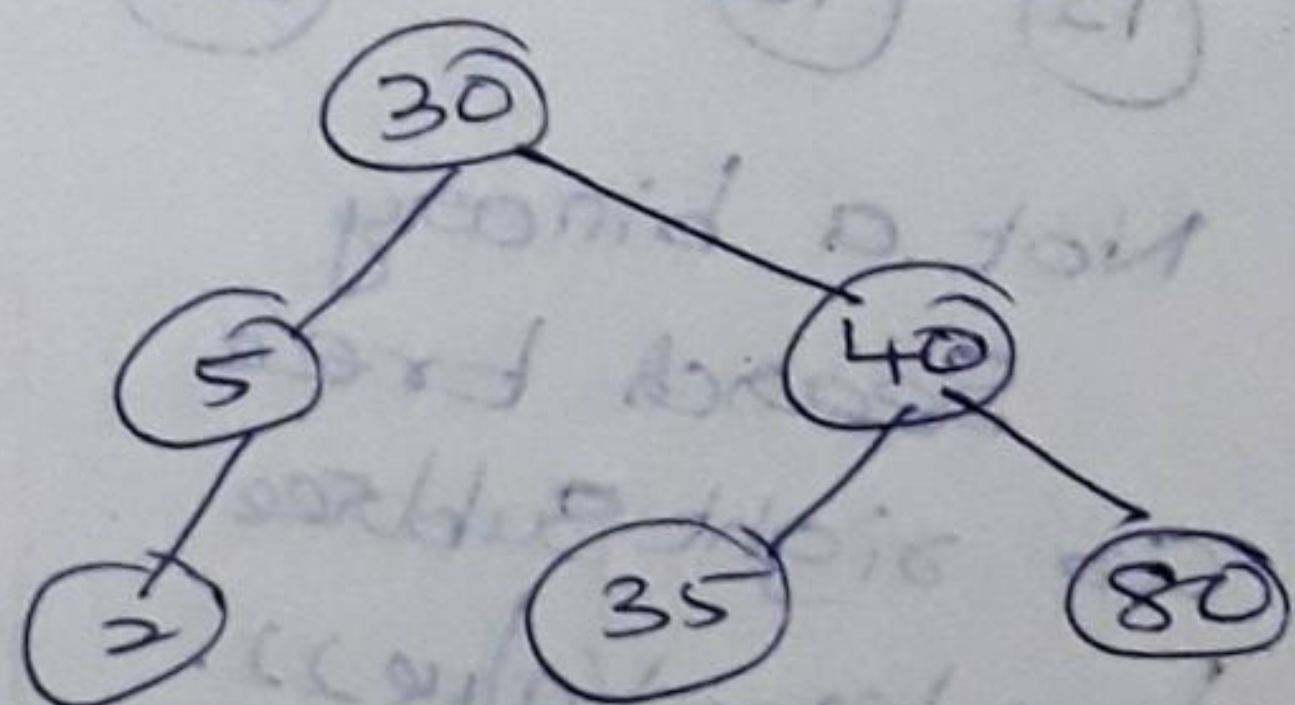
}

② Inserting into a Binary Search Tree :-

\* To insert a pair with key 80 into the tree,



Insert 80



Insert 35

\* The time required to search the tree for  $k$  is  $O(h)$  where  $h$  is the height.

③ Deletion from a binary search tree

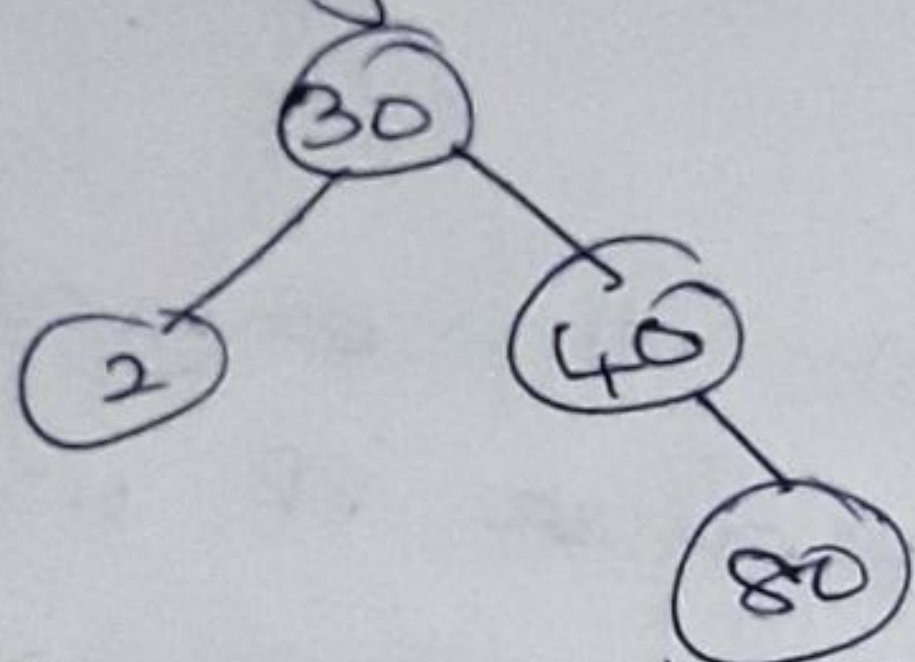
\* To delete 35 from the tree, the left



child field of its parent is set to Zero and the node freed.

\* To delete 80 from this tree, the right child of 40 is set to zero

\* ~~when~~ To delete (5) from the tree, change the pointers from the parent node.



#### [4] Joining and Splitting Binary Trees

##### ① three-way join (Small, mid, big) -

Small is smaller than mid-key

big is greater than mid-key.

Following the join, both Small and big are empty.

##### ② two-way join (Small, big)

all keys of Small are smaller than

all keys of big and join both

Small and big are empty.

##### ③ Split (the Tree, k, Small, mid, big).

The ~~tree~~ is split into three parts: Small is a binary search tree

contains all pairs of the Tree.

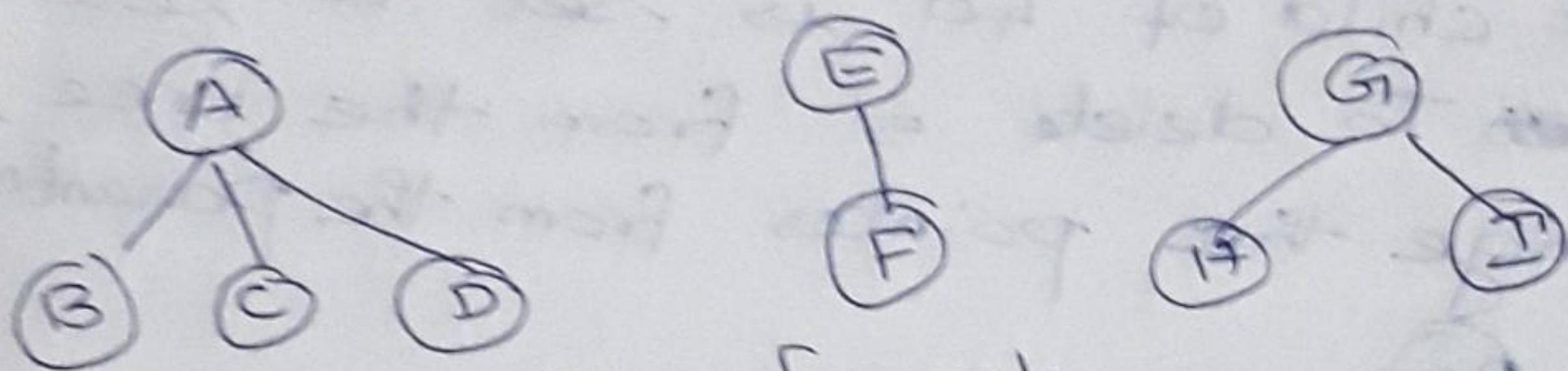
mid is the pair in the Tree.

big is a binary search tree all pairs of Tree that have key larger than k.



## 6. FORESTS

\* A forest is a set of  $n \geq 0$  disjoint trees.

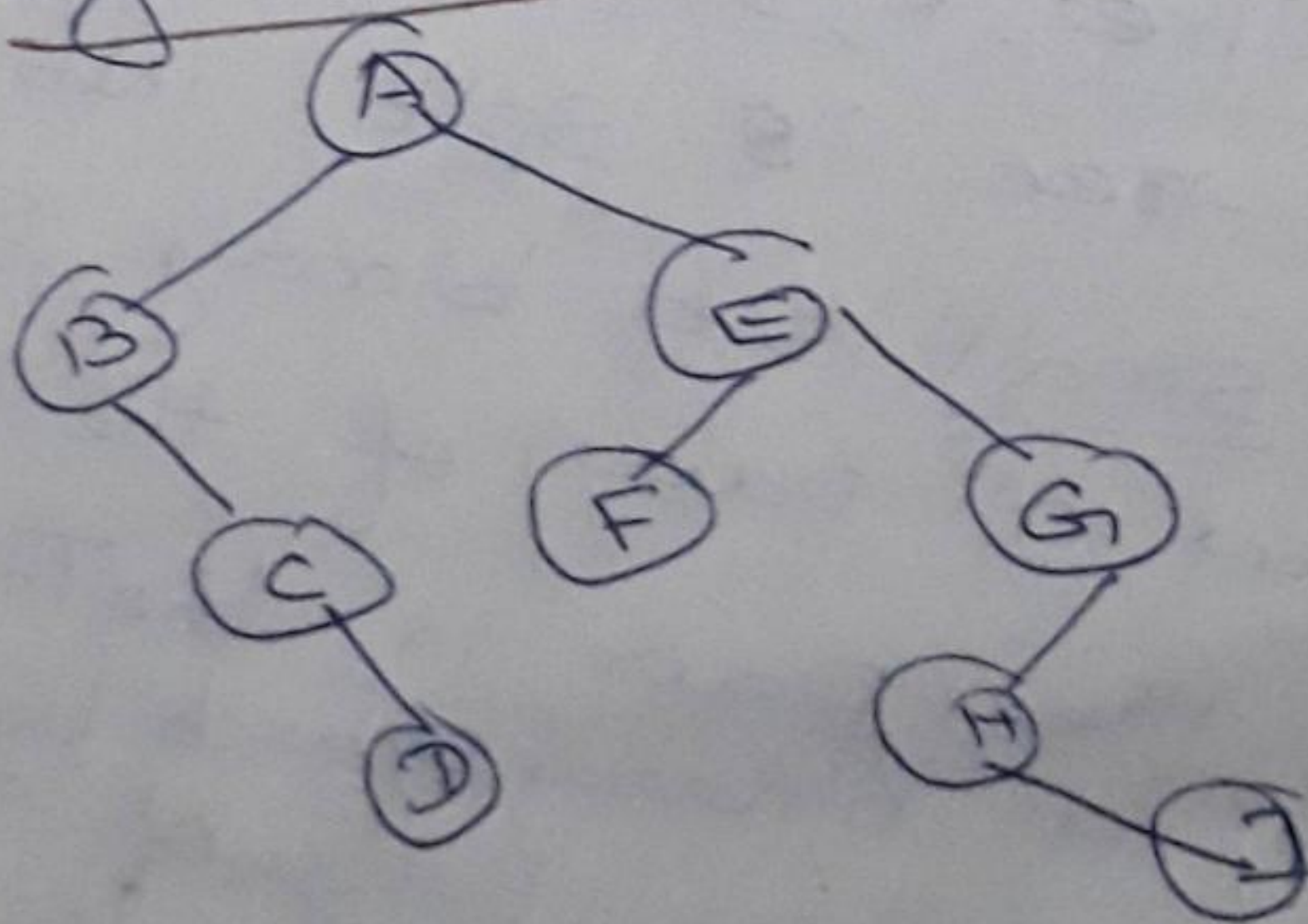


Three-tree forest

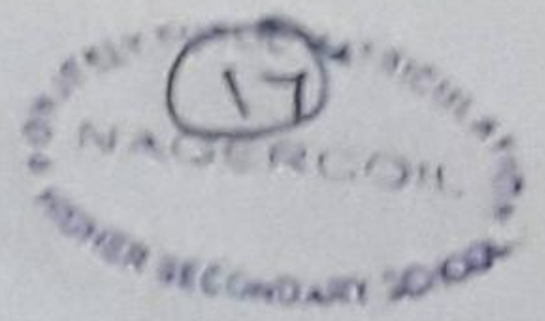
The concept of a forest is very close to that of a tree because if we remove the root of the tree, we obtain a forest.

\* Eg, removing the root of any binary tree produces a forest of 2 trees.

\* To transform a forest into a single binary tree, we obtain the binary tree representation of each of the trees in the forest and then link these binary trees together through the right child field of the root nodes.

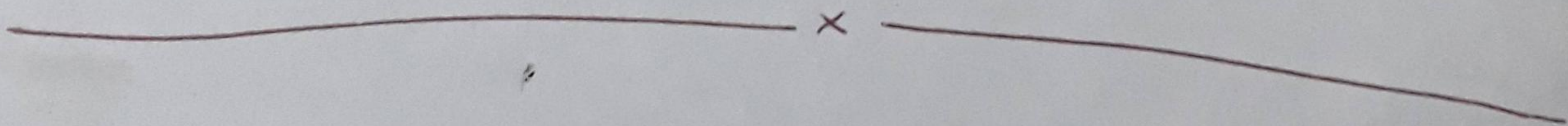






If  $T_1, \dots, T_n$  is a forest of trees, then the binary tree corresponding to this forest denoted by  $B(T_1, \dots, T_n)$

- ① is empty if  $n=0$ .
- ② has root equal to  $\text{root}(T_1)$  has left subtree equal to  $B(T_{i_1}, \dots, T_{i_m})$  where  $T_{i_1}, \dots, T_{i_m}$  are subtrees of  $\text{root}(T_1)$  and has right subtree  $B(T_2, \dots, T_n)$ .





## UNIT - IV

### I. Graph Abstract Datatype

→ Graph is another non-linear data structure.

Graph consists of 2 sets

(1) A set  $V$  called the set of all vertices  $V(G)$

(2) A set  $E$  called the set of all edges  $E(G)$

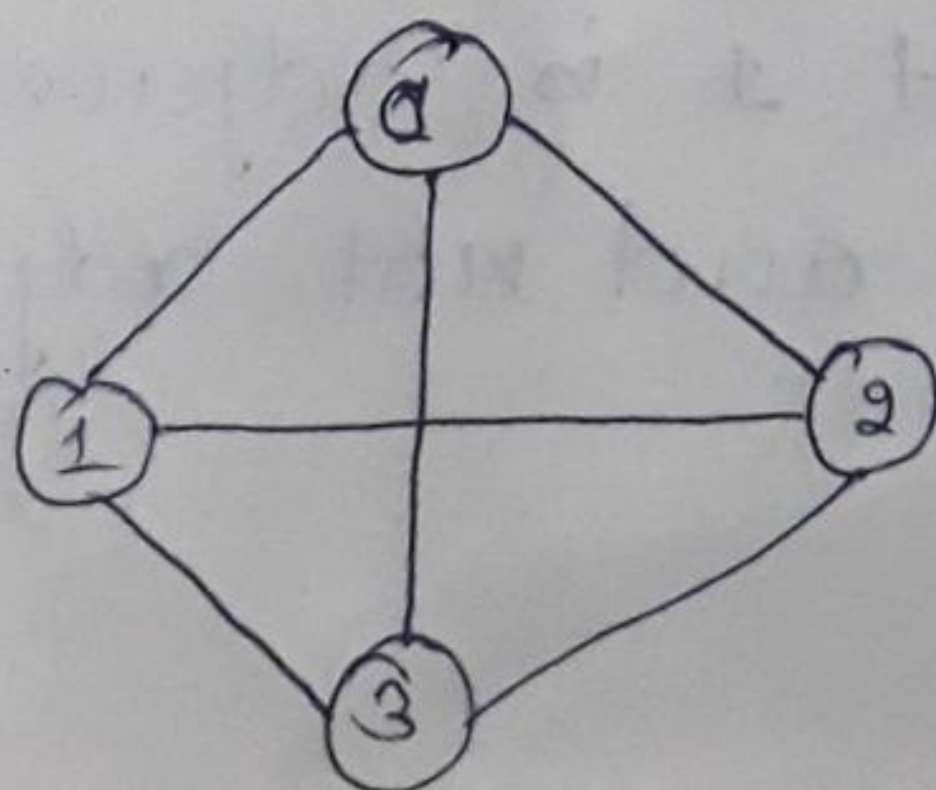
$G=(V,E)$  to represent the set of vertices and edges.

→ In Directed graph each edge is represented by a directed pair  $\langle u,v \rangle$ .  $u$  is the tail &  $v$  is the head.

eg:



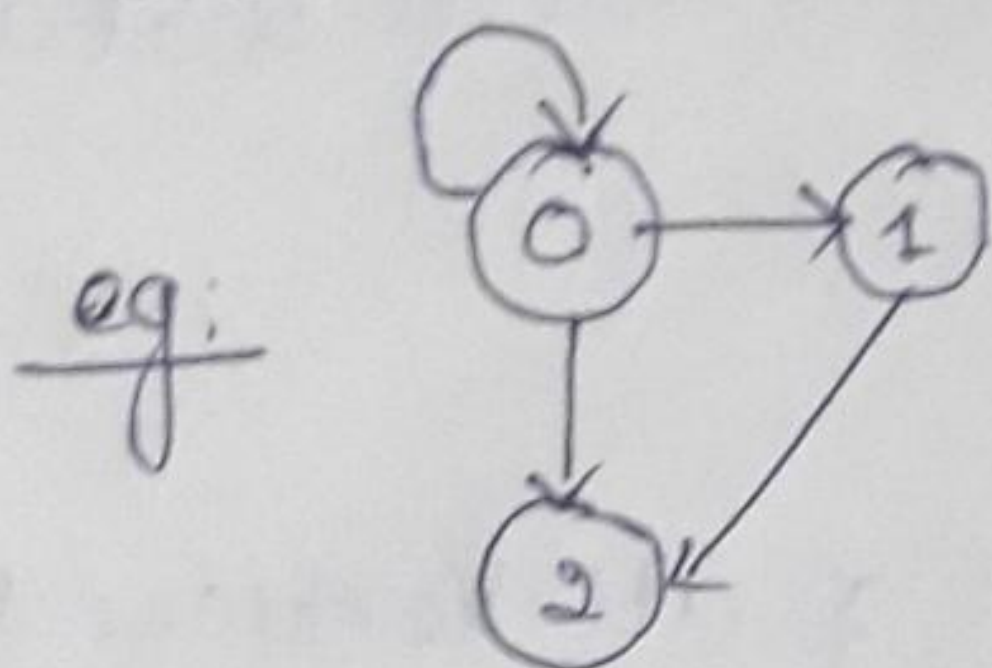
→ In undirected graph a pair of vertices representing any edge is unordered. Thus pairs  $(u,v)$  &  $(v,u)$  represent same edge.





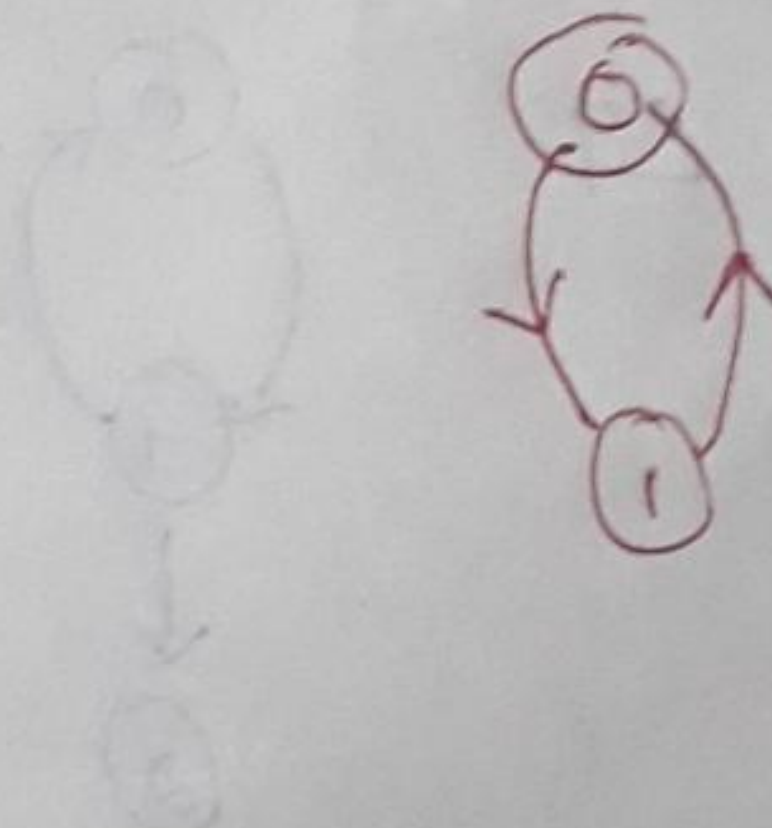
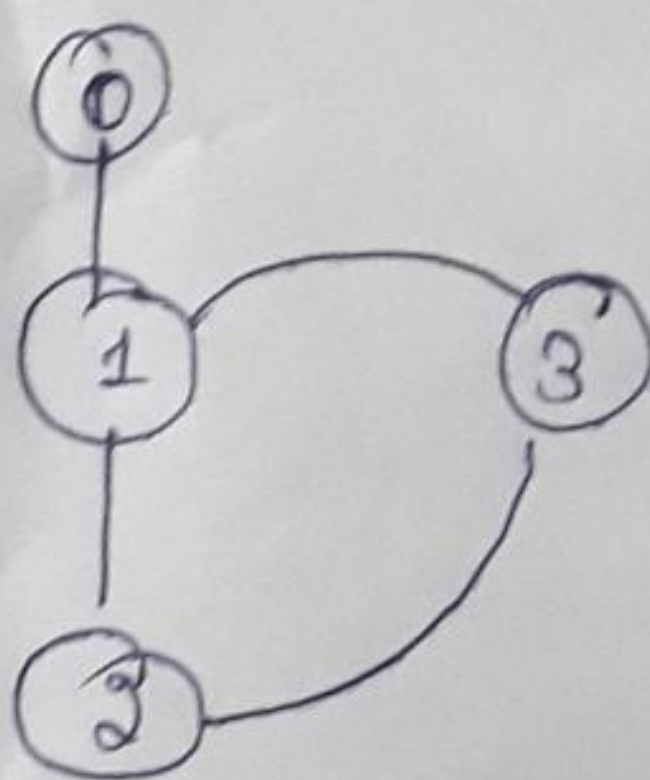
→ self edges or self loop:

If there is an edge whose starting and end vertices are same.



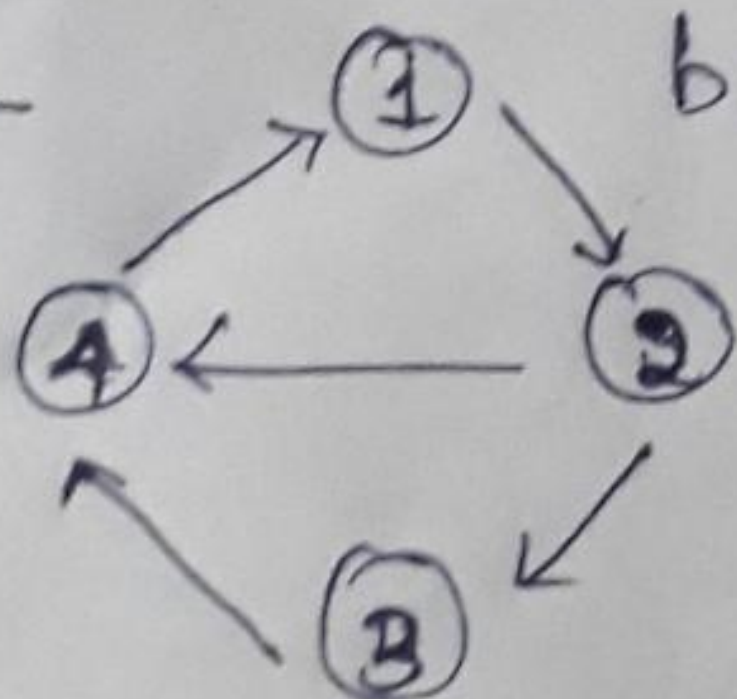
Multiple parallel edges: - If there is more than one edge between the same pair of vertices then they are known as parallel edges. A graph which has either self loop or parallel edges or both is called multigraph.

eg:



Adjacent Vertices: - A Vertex  $u$  is adjacent to another vertex  $v$ , if there is an edge from  $u$  to  $v$ . Here 2 is adjacent to 3 & 4.

eg:-

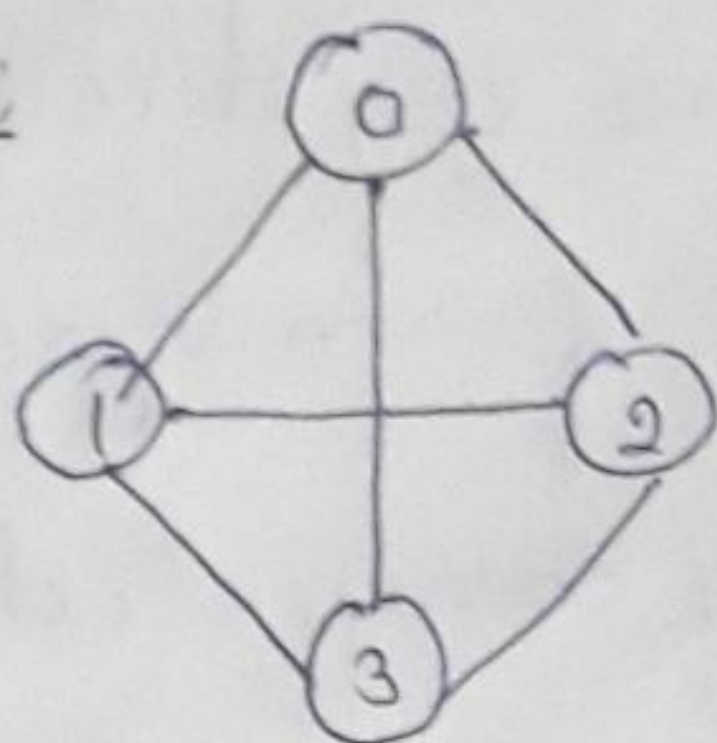


but 1 is adjacent to 2 and not adjacent to 4.



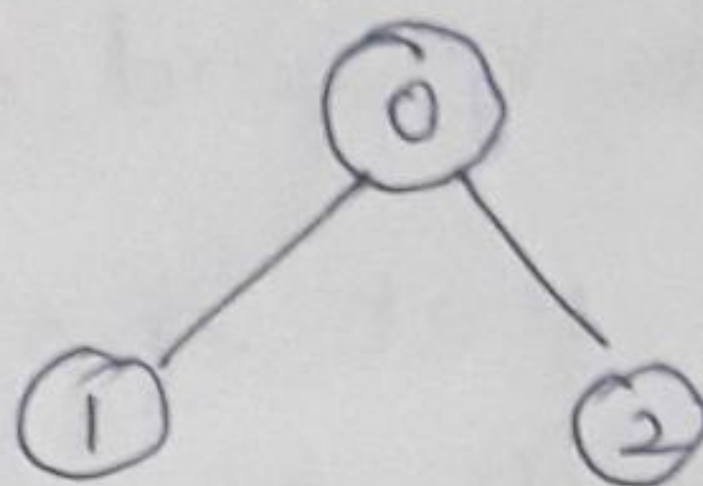
Subgraph:- A subgraph of  $G$  is  $G'$  such that  $V(G') \subseteq V(G)$  &  $E(G') \subseteq E(G)$ .

eg:

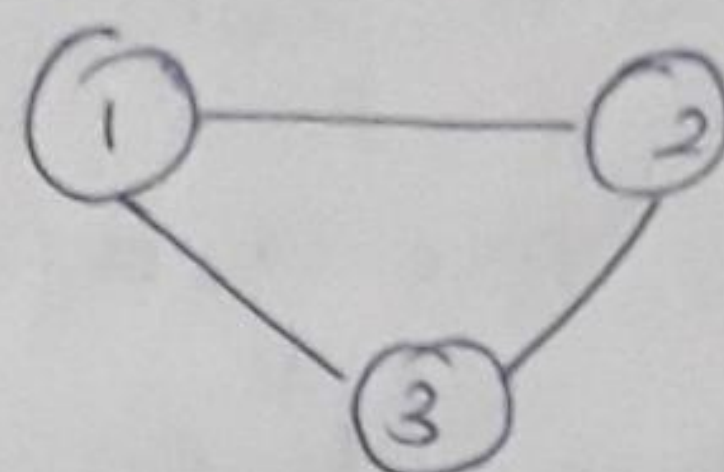


Graph  $G_1$

some of subgraph of graph  $G_1$



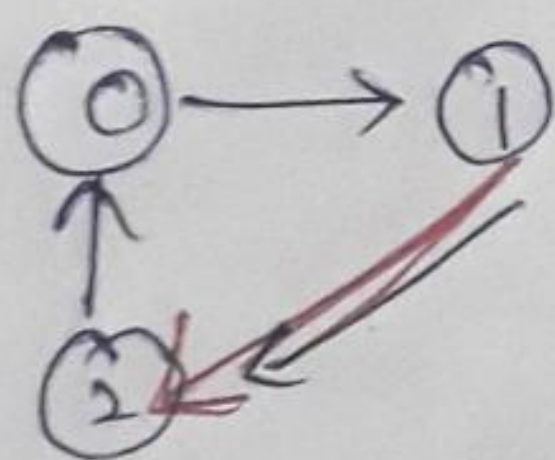
(i)



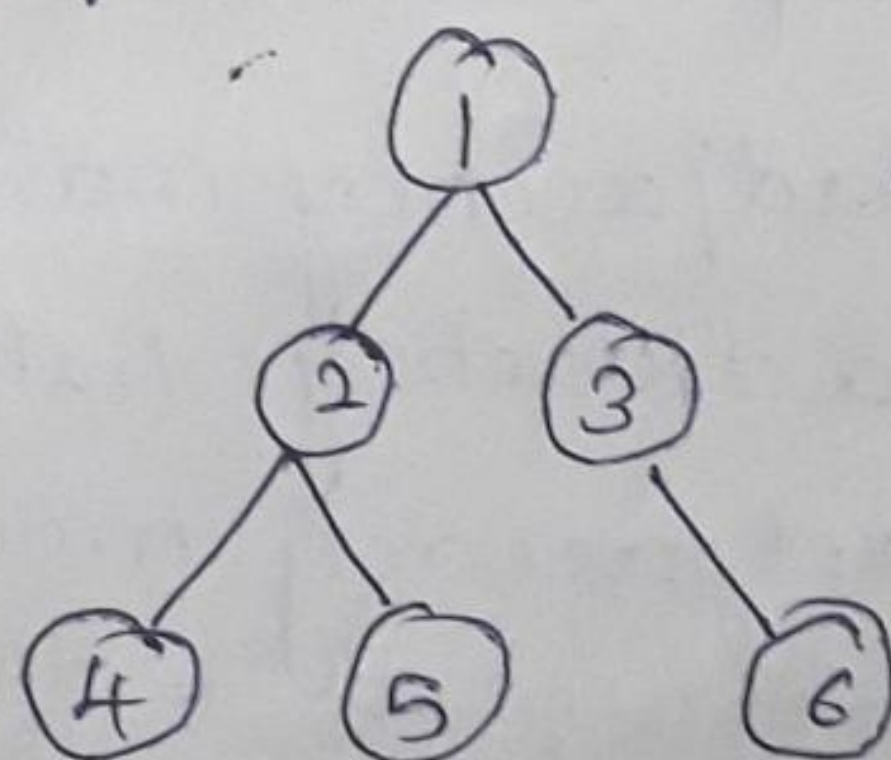
(ii)

cyclic graph: If there is a path containing one or more edges which starts from a vertex and terminates on the same vertex then the path is cycle.

If a graph doesnot have any cycle then it is called acyclic graph.



eg: cyclic graph

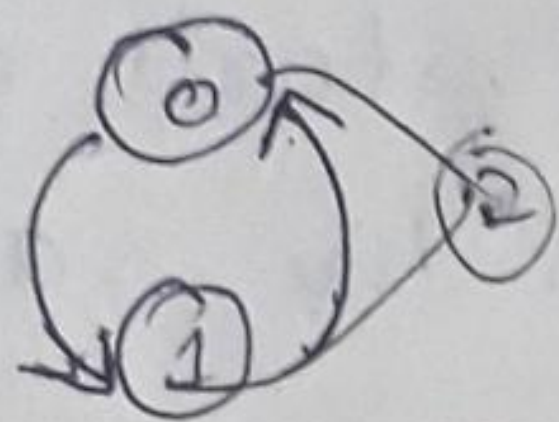


eg: acyclic graph

connected graph: In a graph  $G$ , two Vertices  $V_i$  and  $V_j$  are said to be connected, if there is a path in  $G$  from  $V_i$  to  $V_j$ .  
If for every pair of distinct Vertices  $V_i, V_j$  in  $G$  there is a path.



A graph  $G$  is said to be strongly connected <sup>(1)</sup> if and only if for every pair of distinct vertices  $v_i$  &  $v_j$  in  $V(G)$ , there is a directed path from  $v_i$  to  $v_j$  and also from  $v_j$  to  $v_i$ . If the graph is not strongly connected then it is said to be weakly connected.  
eg. for strongly connected



## 2. Graph representation-

Three mostly ~~used~~ commonly used graph representations are

- (i) adjacency matrices
- (ii) adjacency lists.
- (iii) adjacency multilists.
- (4) Weighted edges.

### 1) adjacency matrices :-

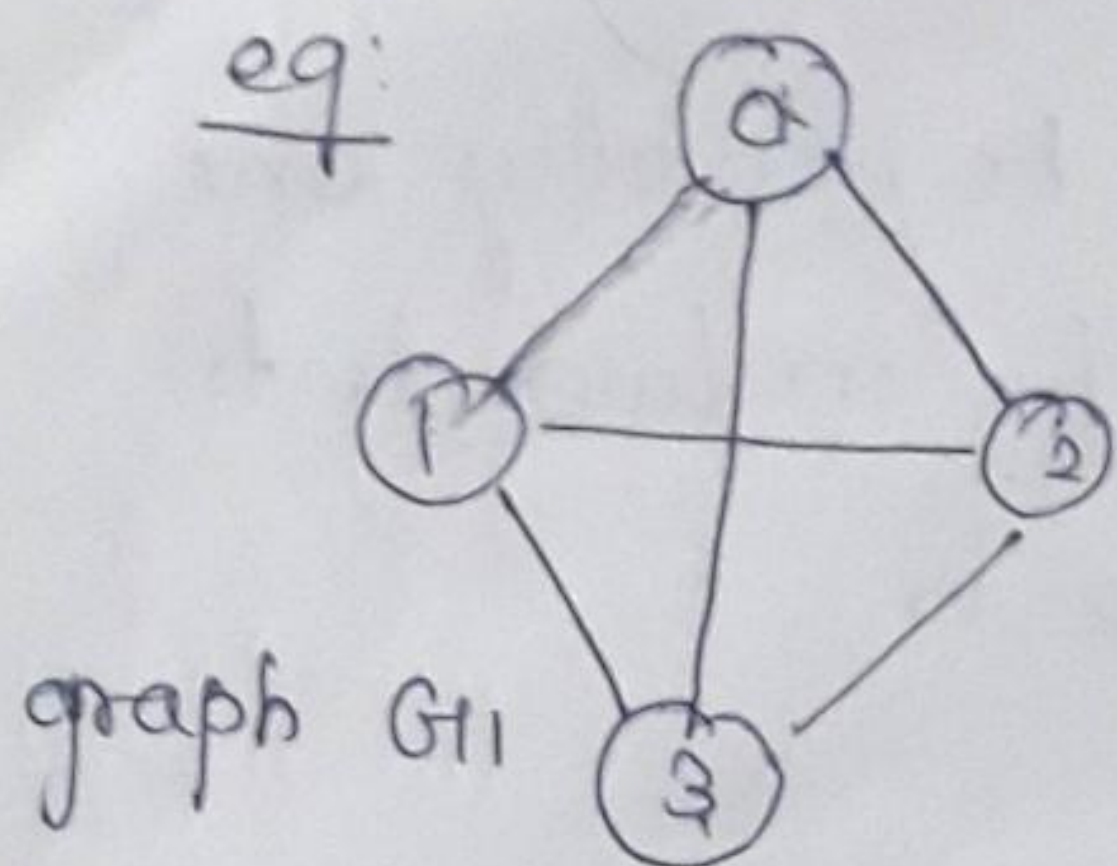
$G = (V, E)$  be a graph with  $n$  Vertices  $n \geq 1$ . The adjacency matrix should be 2-dimensional array  $n \times n$  say 'a'.

$a[i][j] = 1$  if and only if edge  $(i, j)$  is in  $E(G)$ .



$a[i][j] = 0$  iff there is no such edge

eg:

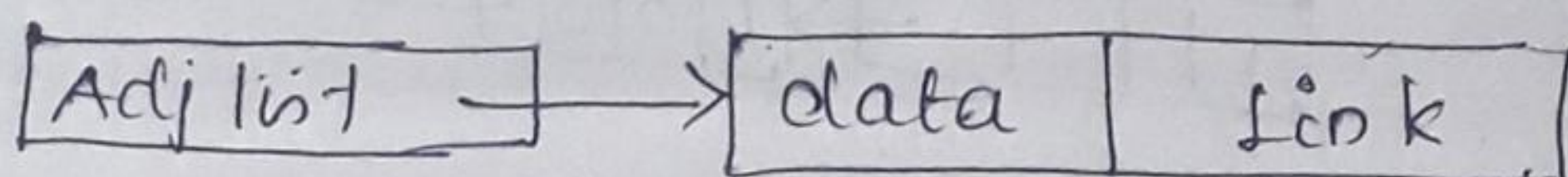


adjacency matrix will be

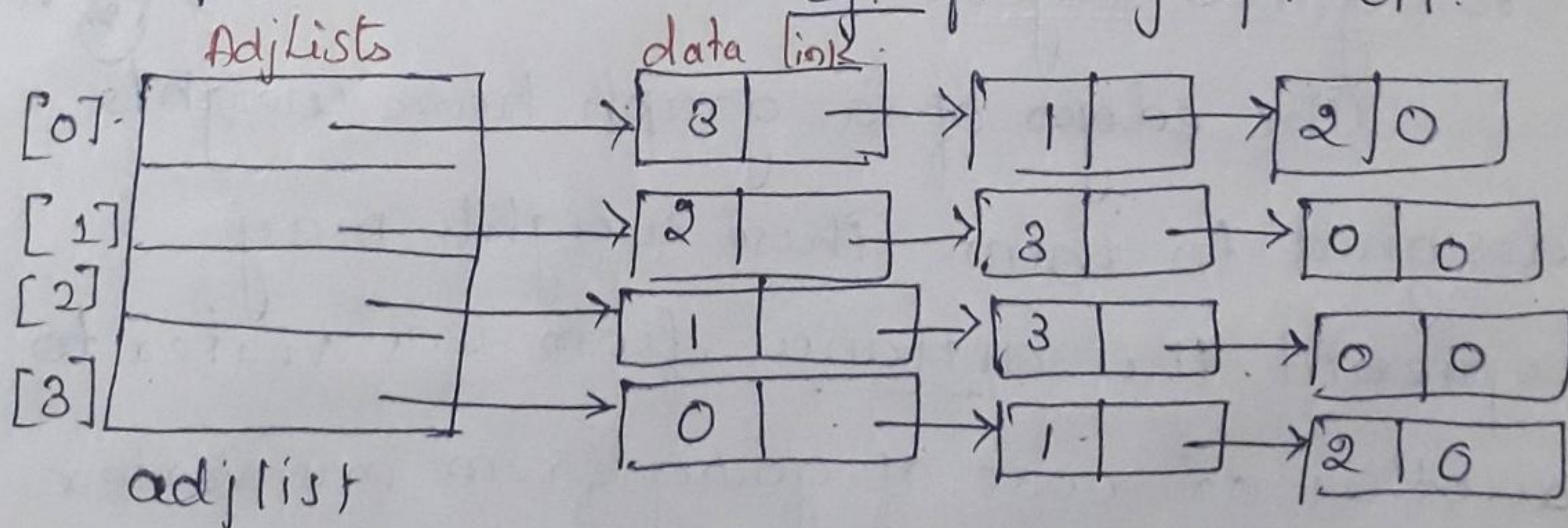
$$\begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

## ② Adjacency lists :-

The  $n$  rows of adjacency matrix are represented as  $n$  chains. There is one chain for each vertex in  $G$ .



data field of an chain stores the index of an adjacent vertex. eg: from graph  $G_1$ .



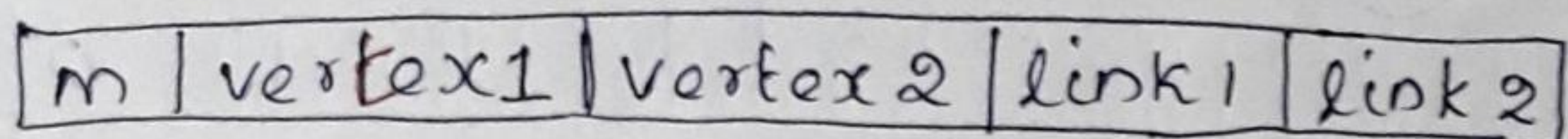
## ③ Adjacency multilists :-

In adjacency list representation of an undirected graph, each edge  $(u,v)$  is

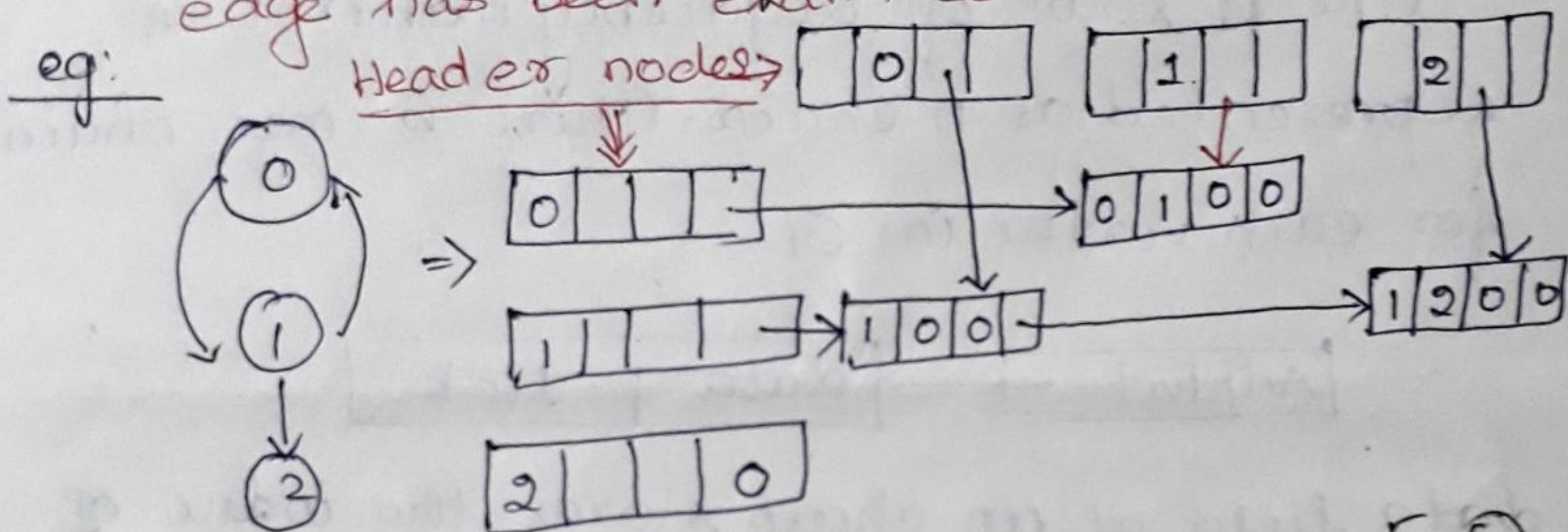


represented by two entries one on the list for u & another list for v. ⑥

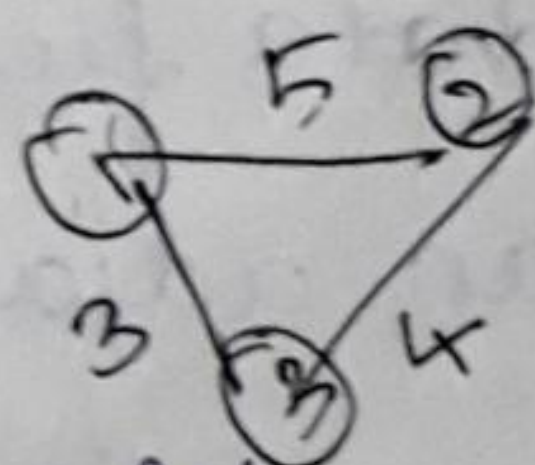
for each edge there will be exactly one node, but this node will be in two lists so the new node structure is



m → boolean mark field whether or not the edge has been examined.



#### ④ Weighted edges:



The edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or cost of going from one vertex to an adjacent vertex.

when adjacency list are used, the weight information kept in additional field, weight. A graph with weighted edge is called a Network.



## II. Elementary Graph Operations: 3.

### a) Depth first Search: -

- Begin the search by visiting the start vertex 'v'.
- select an unvisited vertex 'w' from the v's adjacency list and carryout DFS on w.
- now preserve the current position in v's adjacency list by placing it on a stack.
- Eventually our search reaches a vertex u, that has no unvisited vertices in adjacency list.
- At this point remove a vertex from the stack & continue processing its adjacency list.
- visited vertices are removed from the stack & unvisited vertices are placed in stack.
- Search terminates when stack is empty.
- DFS is similar to preorder tree traversal

```
void dfs (int v)
```

```
{ node pointer w;
```

```
visited [v] = TRUE;
```

```
printf ("%5d", v);
```

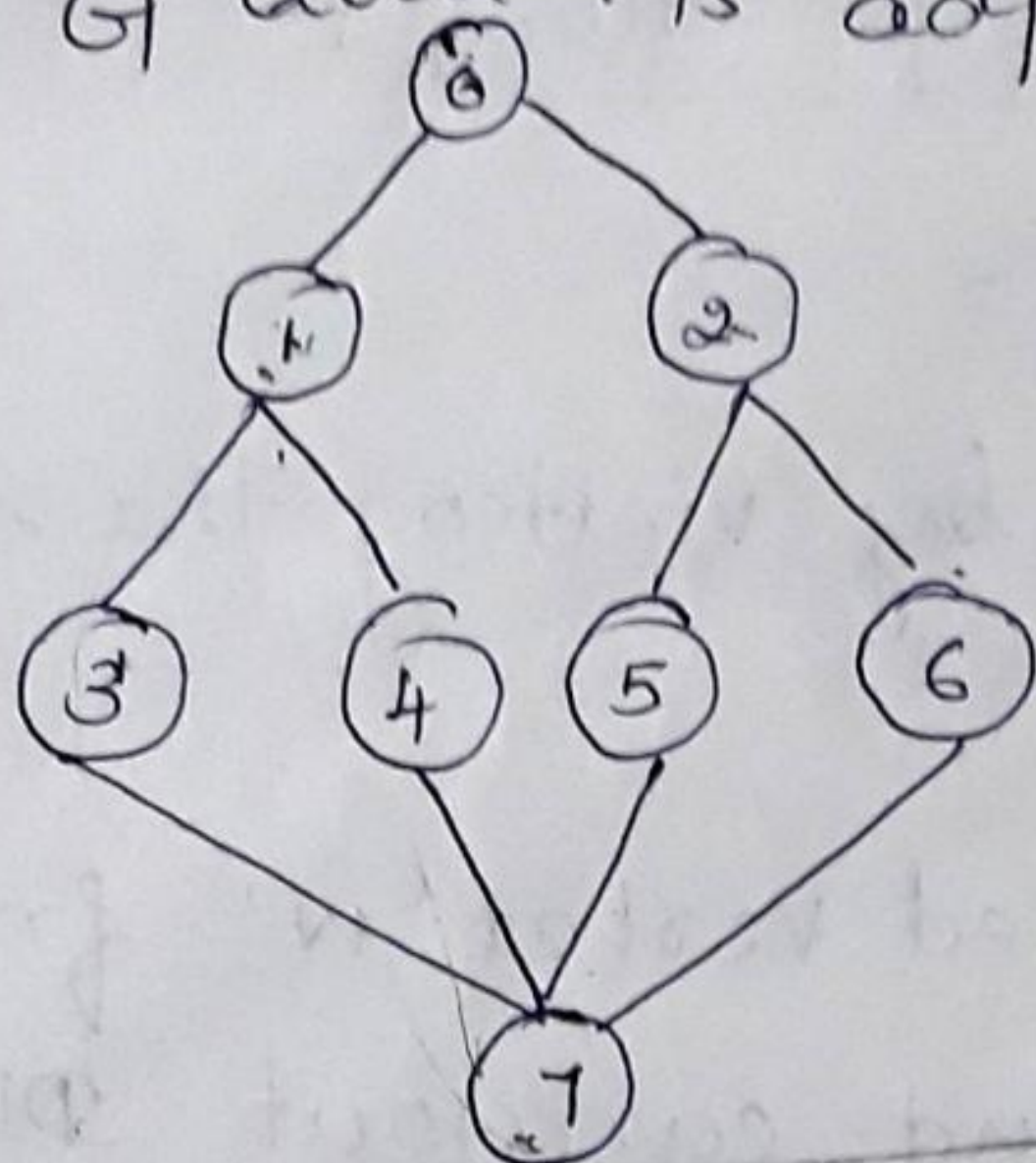
```
for (w = graph[v]; w; w = w → link)
```

```
if (!visited (w → vertex))
```

```
dfs (w → vertex);
```



eg. Graph G and its adjacency list (8)



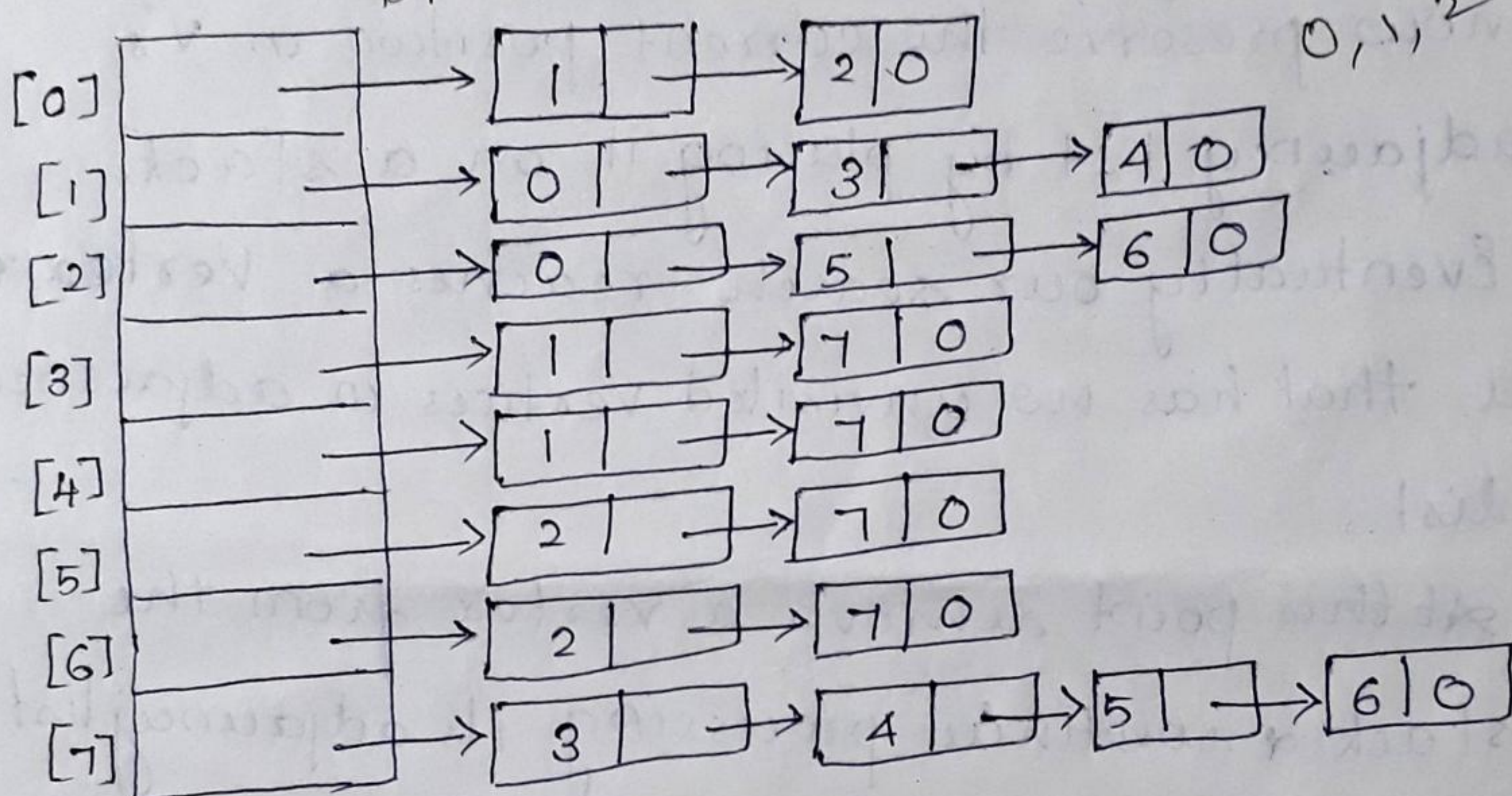
Dfs.  $0 \rightarrow 1 \rightarrow 3 \rightarrow 7$ .

$0 \rightarrow 1 \rightarrow 4 \rightarrow 7$

$0 \rightarrow 2 \rightarrow 5 \rightarrow 7$

$0 \rightarrow 2 \rightarrow 6 \rightarrow 7$

BFS



### b) Breadth First Search:-

→ BFS is similar to level order traversal

→ Begins by at vertex  $v$  & mark it as visited.

→ visit all the vertices in adjacency list.

→ after visiting each vertex place it in a queue.

→ ~~if~~ visiting ~~the~~ each vertex in the adjacency list, remove it from the queue.

→ finished the search, when queue is empty.



void bfs(int v)

{

node pointer w;

front = rear = NULL;

printf ("%5d", v);

visited[v] = TRUE;

addq(v);

while (front) {

    v = deleteq();

    for (w = graph[v]; w = w → link)

        if (!visited[w → vertex])

        { printf ("%5d", w → vertex);

          addq(w → vertex);

          visited[w → vertex] = TRUE;

        }

    }

}

c) connected components :-

→ we can implement this operation by simply calling either dfs(0) or bfs(0) and then determining if there any unvisited vertices.



void connected (void)

(6)

```
{ int i;  
  for (i=0; i<n; i++)  
    if (!visited[i])  
      { dfs(i);  
        printf("\n");  
      }  
}
```

### d) Spanning Trees :-

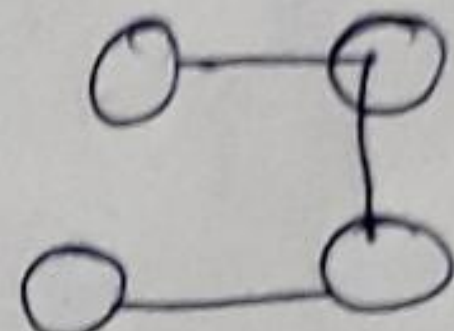
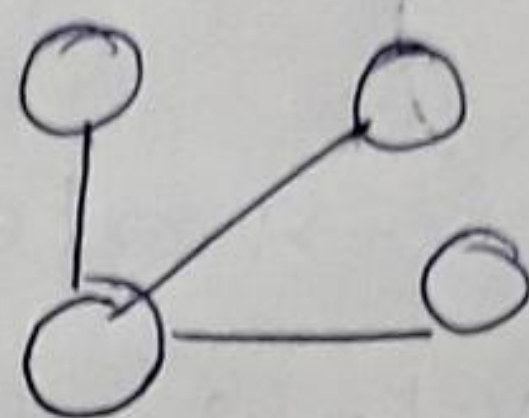
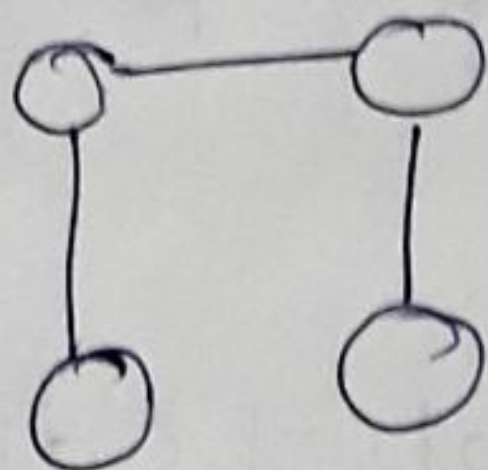
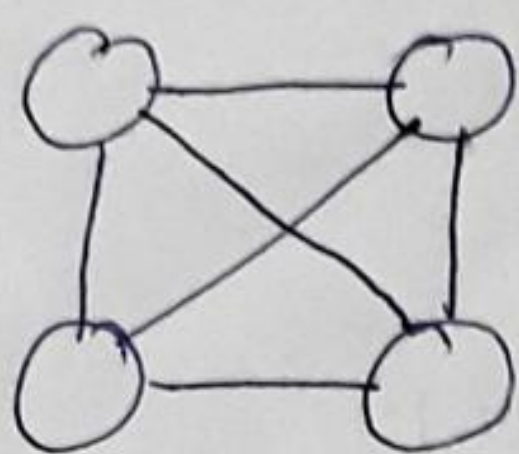
- graph  $G$  is connected, a depth first or breadth first search starting at any vertex visits all the vertices  $G$ .
- partition the edges  $G$  into two sets
  - \*  $T$  for tree edges +  $N$  for non tree edges.
- $T$  edges used for traversed during searches +  $N$  is the set of remaining edges.
- Spanning Tree is any tree that consists solely of edges in  $G$  and that includes all the vertices in  $G$ .
- dfs or bfs to create a spanning tree.



→ when dfs is used, resulting spanning tree is known as depth first spanning tree

→ when bfs is used, resulting spanning tree is known as breadth first spanning tree.

eg:



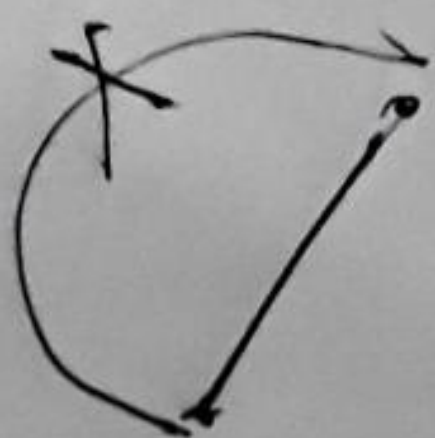
A complete graph & three of its spanning trees.

② Biconnected components:—

→ An articulation point is a vertex  $v$  of  $G$  such that the deletion of  $v$ , together with all edges incident on  $v$ , produces a graph  $G'$  that has at least two connected components.

→ Biconnected graph is a connected graph that has no articulation points.

→ A biconnected component of a connected undirected graph is a maximal biconnected subgraph  $H$  of  $G$ . By maximal, we mean  $G$  contains no other subgraph that is both biconnected and properly contains  $H$ .





void bicon(int u, int v)

(12)

{  
  nodepointer ptr;

  int w, x, y;

  dfn[u] = low[u] = num++;

  for (ptr = graph[u]; ptr; ptr = ptr->link)

  {

    w = ptr->vertex;

    if (v != w && dfn[w] < dfn[u])

      push(u, w);

      if (dfn[w] < 0)

        bicon(w, u);

      low[u] = MIN2(low[u], low[w]);

      if (low[w] >= dfn[u])

      {

        printf("New biconnected component"),

        do

        {

          pop(x, y);

          printf("<%d, %d>", x, y);

        }

        while (!(x == u && y == w));

        printf("\n");

      }

  }

  else if (w != v) low[u] = MIN2(low[u], dfn[w]);

  }

}



### III. Minimum Cost Spanning Trees ~~4~~.

• A minimum cost spanning tree is a spanning tree of least cost. Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph. algorithms were

- \* kruskal's algorithm
- \* prim's algorithm
- \* ~~sollin's algorithm~~

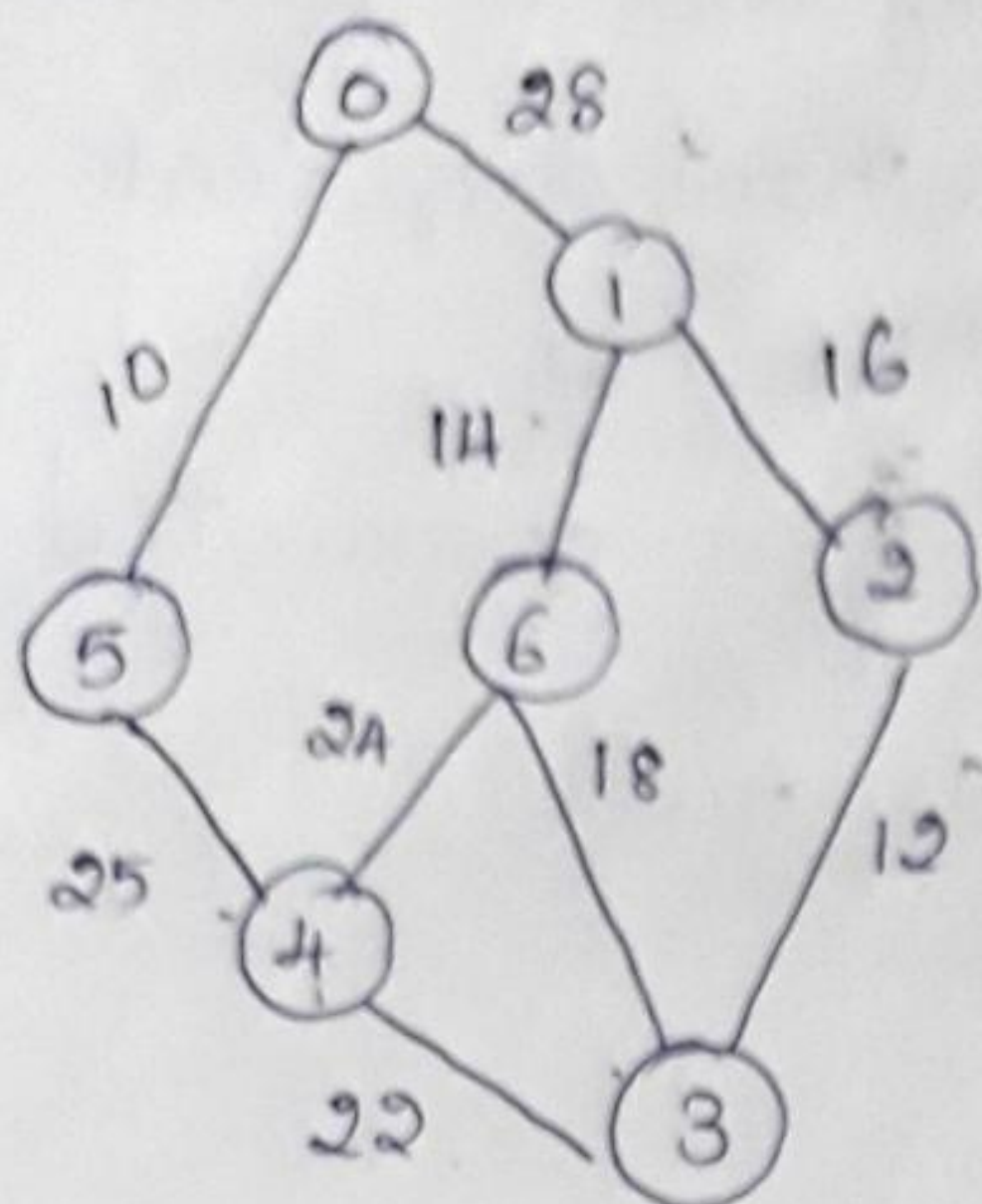
#### ① kruskal's algorithm:

- (i) List all the edges of the graph  $G$  in the increasing order of weights.
- (ii) select the smallest edge from the list and add it into the spanning tree, if the inclusion of this edge doesnot make a cycle.
- (iii) If the selected edge with smallest weight forms a cycle, remove it from the list.
- (iv) Repeat steps 2 to 3 until the tree contains  $n-1$  edges or list is empty.
- (v) If the tree  $T$  contains less than  $n-1$  edges & the list is empty, no spanning tree is possible for graph. else return.

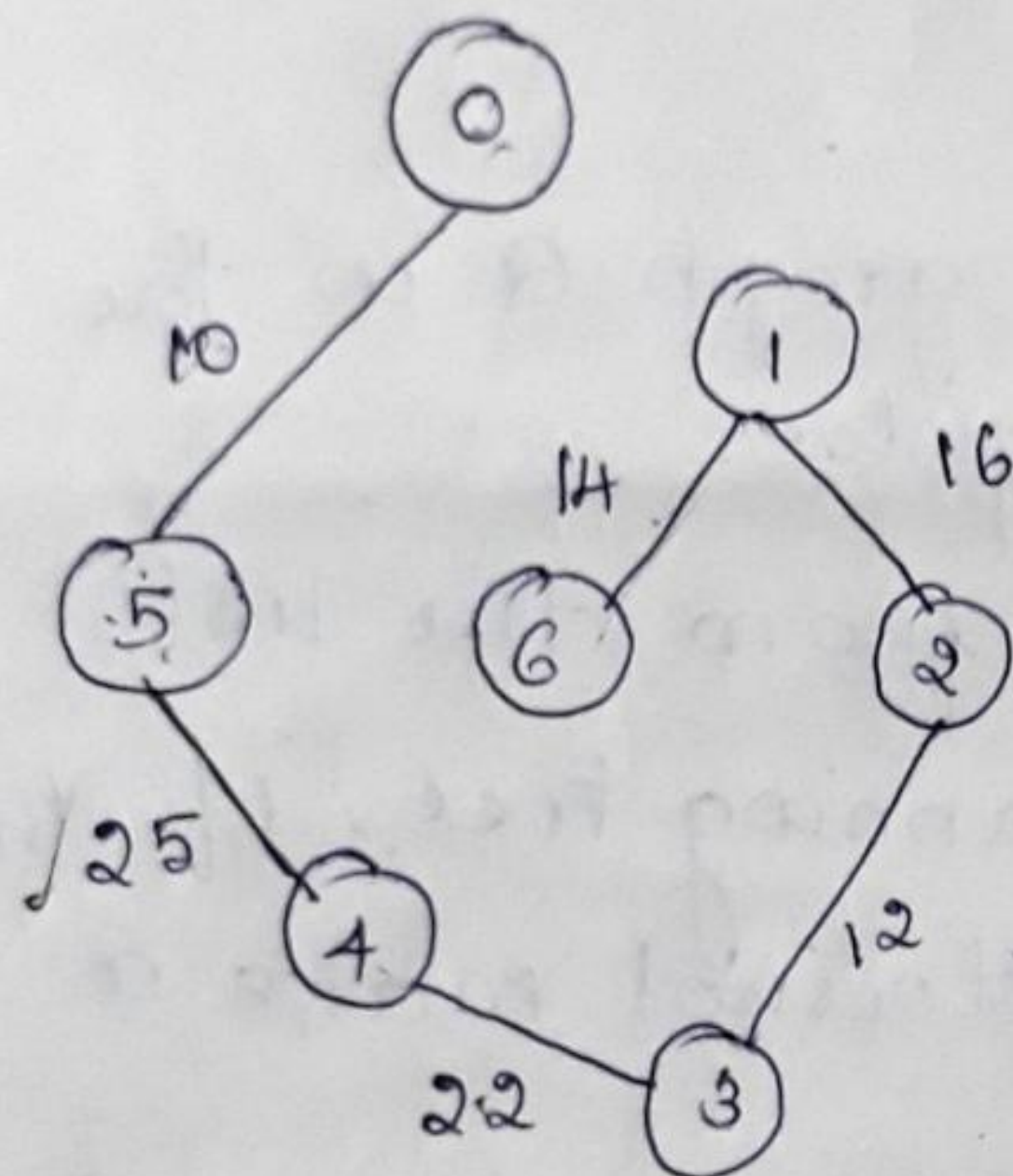


# minimum spanning tree

(12)



⇓



Edge	weight	Result
(0,5)	10	added
(2,3)	12	added
(1,6)	14	added
(1,2)	16	added
(3,6)	18	discarded
(3,4)	22	added
(4,6)	24	discarded
(4,5)	25	added
(0,1)	28	not considered

$T = \{ \}$

while ( $T$  contains less than  $n-1$  edges &  $E$  is not empty)

{  
 choose a least cost edge  $(v, w)$  from  $E$ ;  
 delete  $(v, w)$  from  $E$ ;  
 if  $((v, w)$  does not create a cycle in  $T$ )  
   add  $(v, w)$  to  $T$ ;  
 else  
   discard  $(v, w)$ ;  
 }

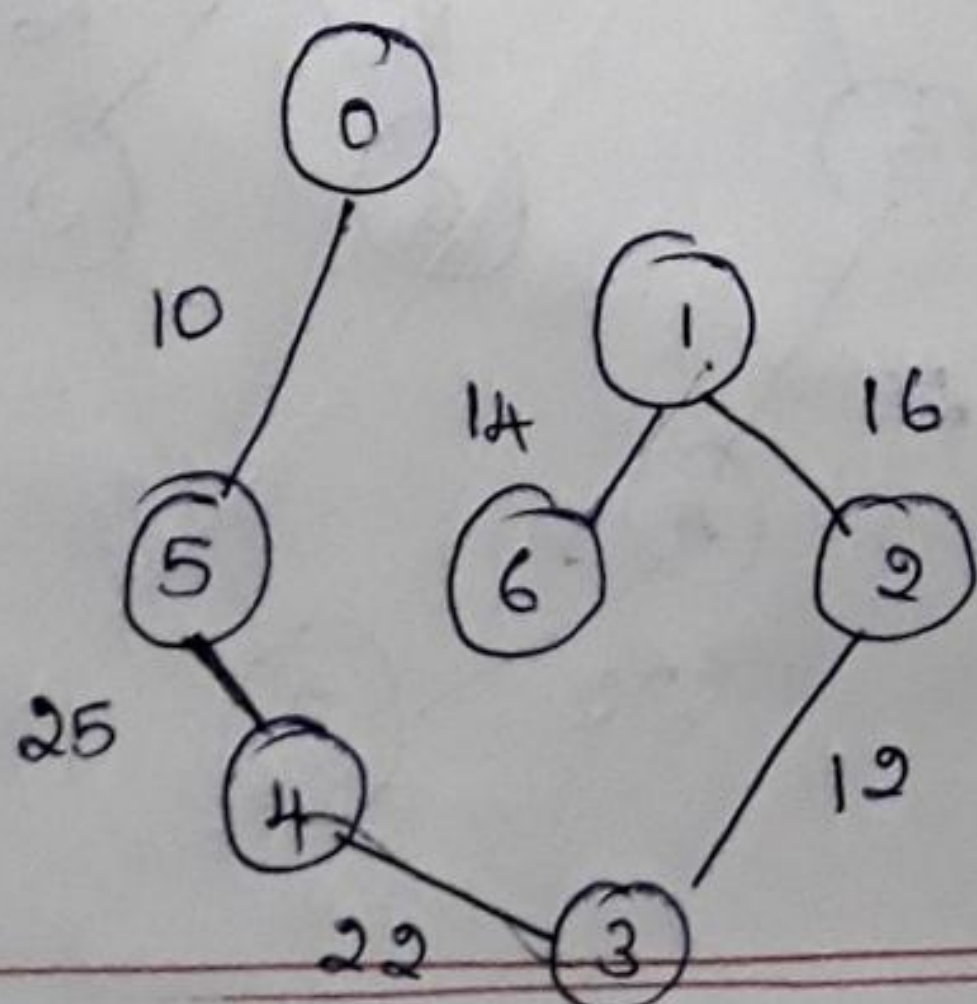
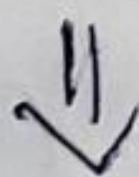
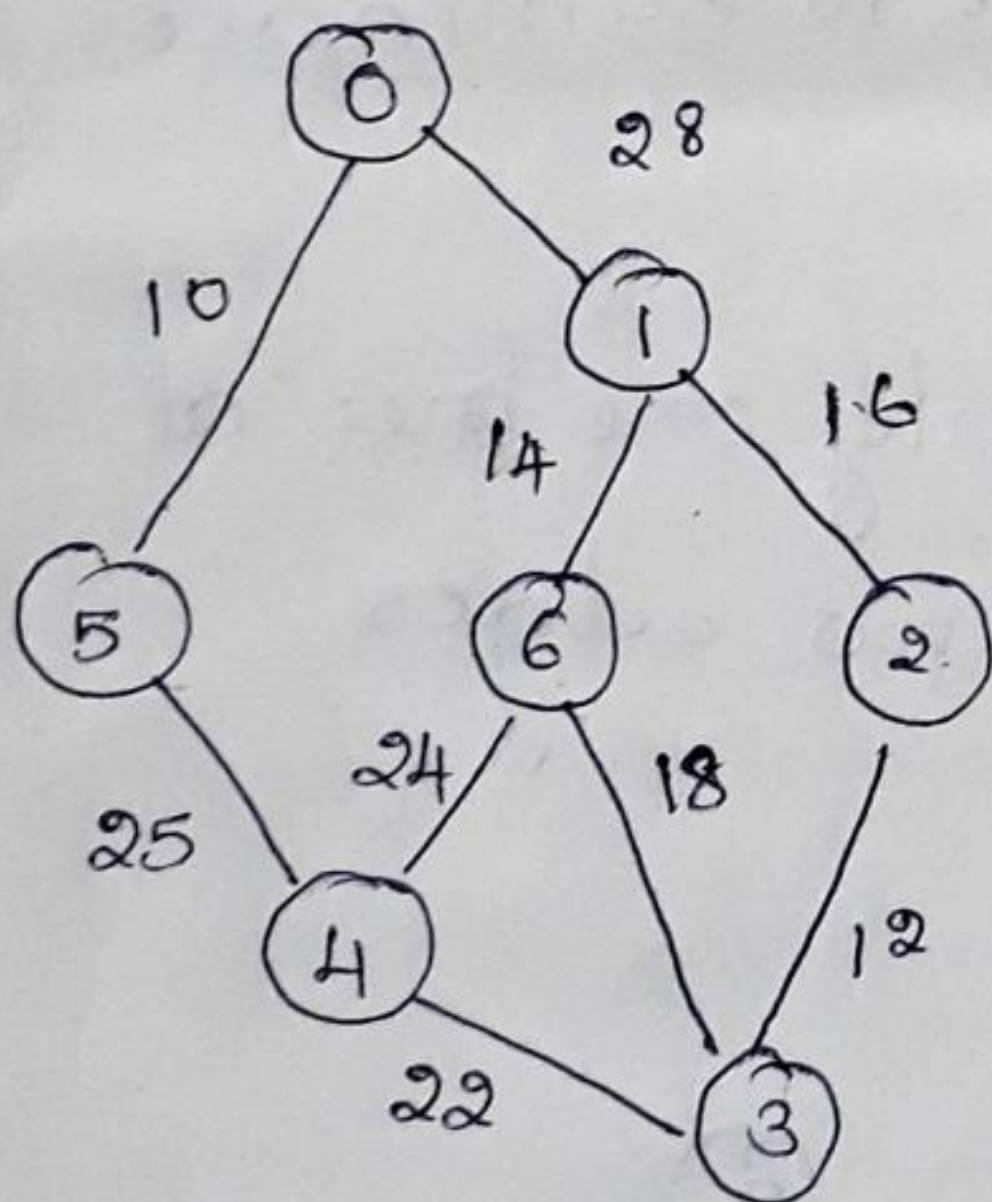


if (T contains fewer than  $n-1$  edges)  
 printf ("No spanning tree (n)");

b) prim's algorithm:-

A minimum spanning tree grows in successive stages. Only a set of vertices included in the tree, rest of the vertices have not.

finds a new vertex to add it to the tree by choosing the edge  $\langle v_i, v_j \rangle$  the smallest among all the edges.



	0	1	2	3	4	5	6	7
0	-	28	-	-	-	(10)	-	-
1	28	-	16	-	-	-	(14)	-
2	-	16	-	(12)	-	-	-	-
3	-	-	12	-	(22)	-	18	-
4	-	-	-	22	-	(25)	24	-
5	10	-	-	-	(25)	-	-	-
6	-	(14)	-	18	24	-	-	-
7	-	-	-	-	-	-	-	-



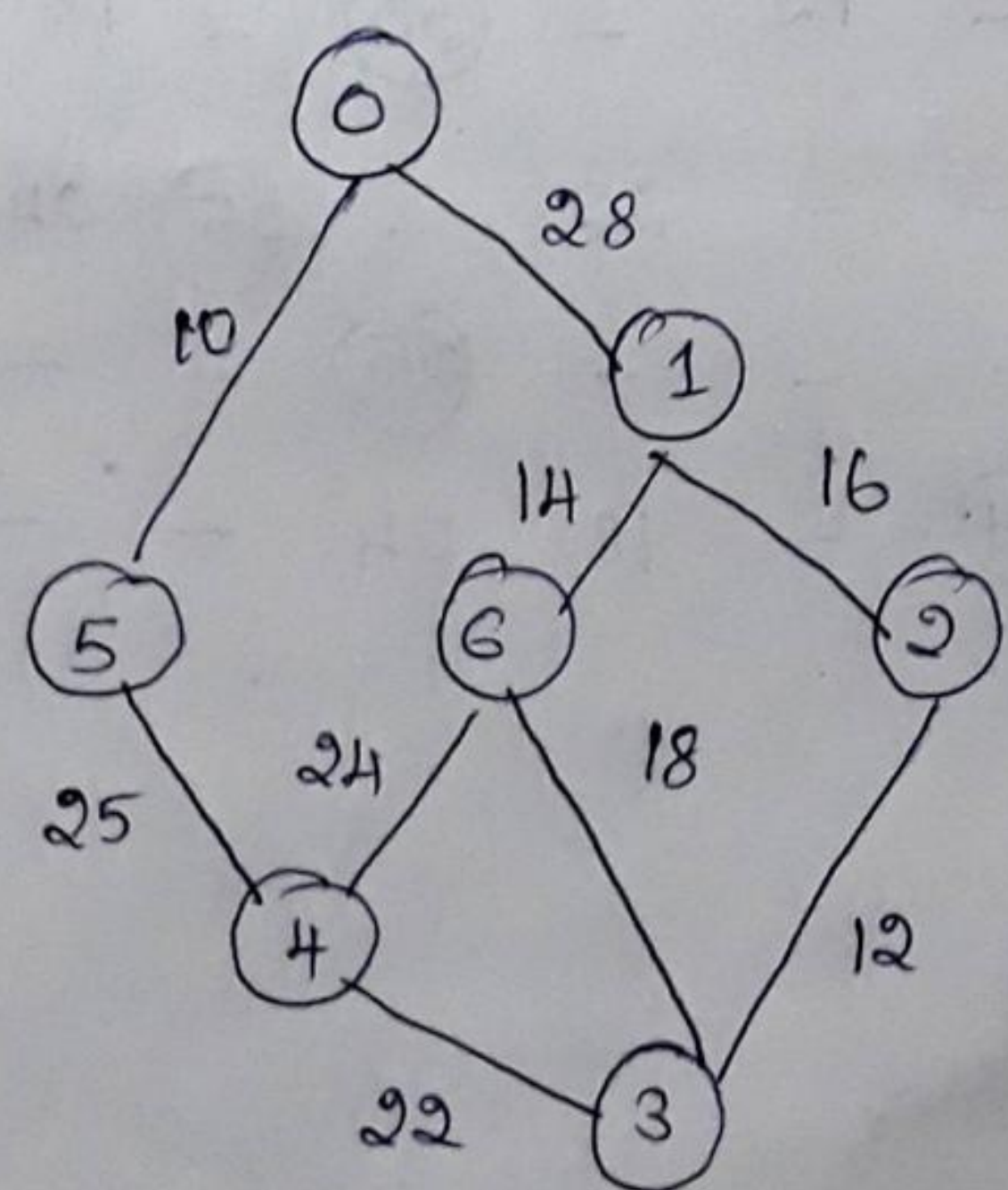
# Sollin's algorithm :-

- sollin's algorithm selects several edges for inclusion in T at each stage
- At the start of a edges, together with all n graph vertices form a spanning forest.
- During a stage, we select one edge for each tree in the forest

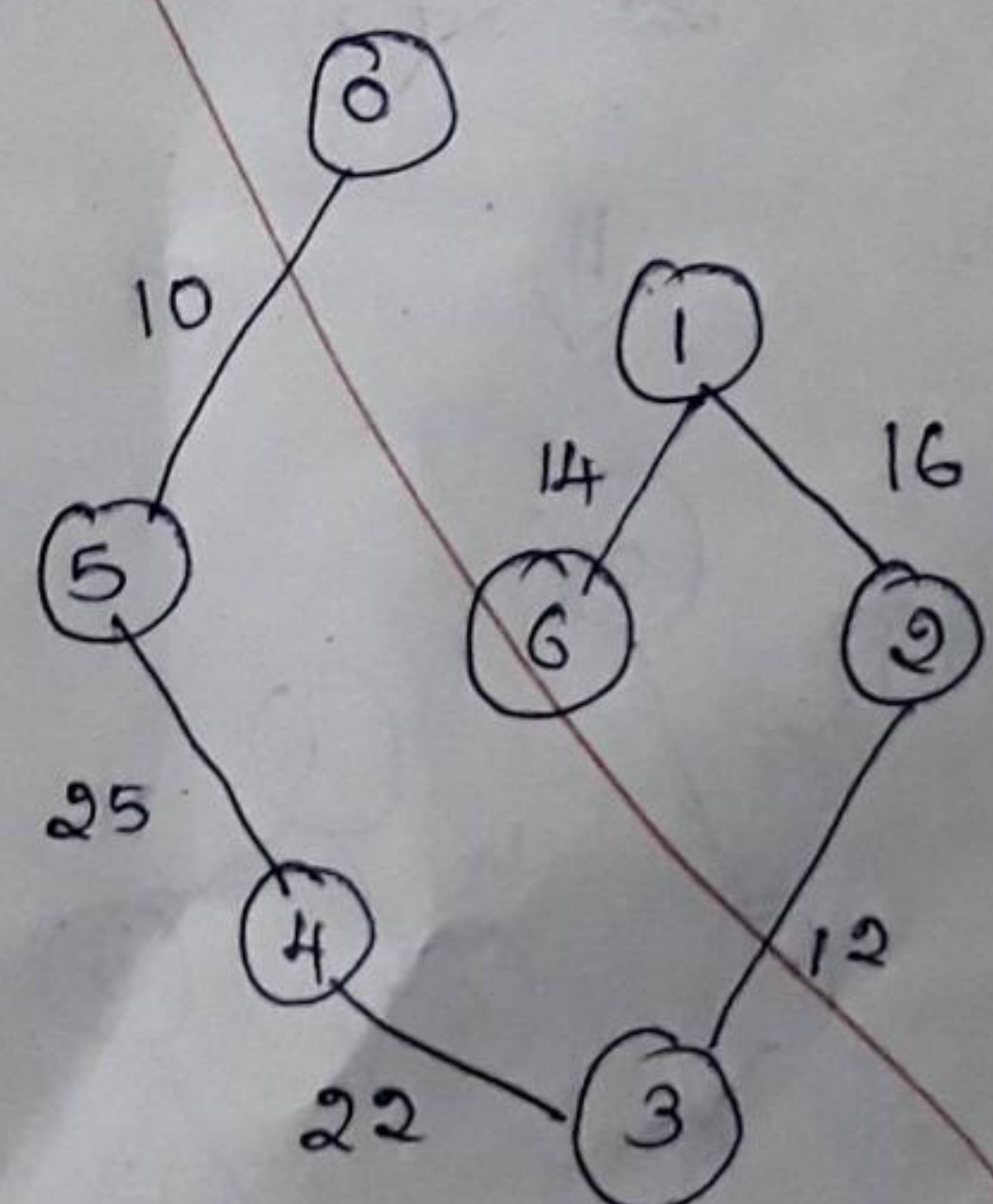
This edge is a minimum cost edge that has exactly one vertex in the tree

- since, two trees in the forest could select the same edge, we need to eliminate multiple copies of edge

- algorithm terminates only one tree at the end of a stage or no edges remain for selection



⇒





## Shortest <sup>5.</sup> paths and Transitive closure: (17)

a) Single Source / All Destinations: Non-negative Edge costs

→ In a directed graph  $G = (V, E)$  a weighting function  $w(e)$ ,  $w(e) \geq 0$  for the edges of  $G$  and a source vertex  $v_0$ , to determine a shortest path from  $v_0$  to each of the remaining vertices of  $G$ .

The paths in non-increasing order of length leads to the following observations

(1) If the next shortest path is to vertex  $u$  then the path from  $v_0$  to  $u$  goes through only those vertices that are in  $S$ . we must show all the intermediate vertices on the shortest path from  $v_0$  to  $u$  is already in  $S$ . assume vertex  $w$  on this path that is not in  $S$ .

\* Then the path from  $v_0$  to  $u$  also contains a path from  $v_0$  to  $w$  which has a length that is less than the length of the path from  $v_0$  to  $u$ .

\* shortest path generated in the non-decreasing order.



(2) vertex  $u$  is chosen so that it has the (18)  
minimum distance,  $\text{distance}[u]$  among all  
the vertices not in  $S$ .

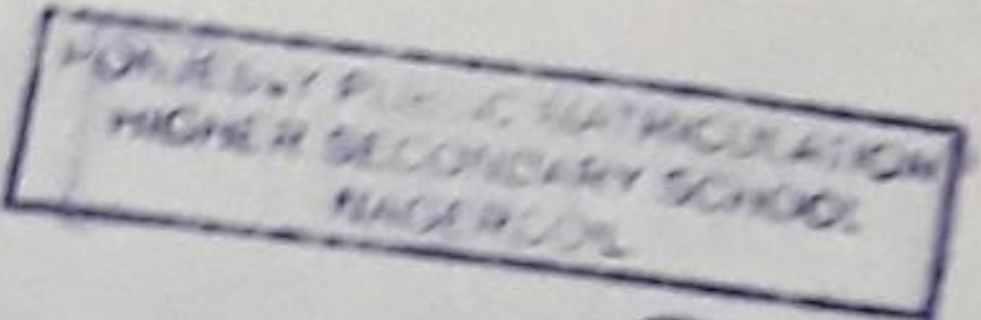
If there are several vertices not in  $S$   
with the same distance then we may  
select any one of them.

(3) once  $u$  is selected and generate the  
shortest path from  $V_0$  to  $u$ ,  $u$  becomes the  
member of  $S$ . Adding  $u$  to  $S$  can change  
the distance of shortest paths starting  
at  $V_0$  going through vertices only in  $S$   
and ending at a vertex  $w$ , which is  
not currently in  $S$ .

The length of shorter path is

$$\text{distance}[u] + \text{length}(\langle u, w \rangle)$$




  
 30/8

void shortest path (int v,   
     int cost[][MAX-VERTICES],   
     int distance[], int n, short int found[])

```

{
    int i, u, w;
    for (i = 0; i < n; i++)
    {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i = 0; i < n - 2; i++)
    {
        u = choose (distance, n, found);
        found[u] = TRUE;
        for (w = 0; w < n; w++)
        {
            if (!found[w])
            {
                if (distance[u] + cost[u][w] < distance[w])
                {
                    distance[w] = distance[u] + cost[u][w];
                }
            }
        }
    }
}
  
```



Sorting 1.

→ List means collection of records, each records contain one or more fields. The fields used to distinguish among the records know as keys.

→ The datatype of each record is element and each record is assumed to have an integer field key.

```
int seqSearch (element a[], int k, int n)
```

```
{ int i;
```

```
  for (i = 1; i <= n && a[i].key != k; i++);
```

```
  if (i > n) return 0;
```

```
  return i;
```

```
}
```

program: Sequential Search.

~~One way to be~~

→ Searching a record with a ~~the~~ specified key is to examine the list of records in left to right or right to left order such a order is known as sequential search.



②  
\* for example consider list 1 be the employer list and list 2 be employee list. Let  $\text{list}[i].\text{key}$  and  $\text{list2}[i].\text{key}$  respectively denote the key of the  $i$ th record in list 1 & list 2.  
\* we make the following assumptions about the required verification.

(1) If there is no record in the employee list corresponding to a key in the employer list, a message is to be sent to the employee.

(2) If reverse is true then the message is sent to the employer.

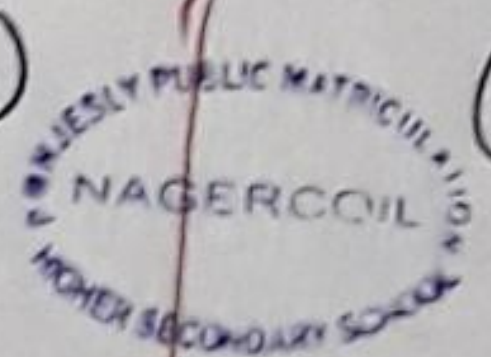
(3) If there is a discrepancy between two records with the same key, a message to this effect is to be output.

### Insertion Sort:-

\* This method is to insert a new record into a sorted sequence of  $i$  records in such a way that the resulting sequence of size  $i+1$  is also ordered.



void insert(element e, element a[], int i)



3

```
{ a[0] = e;
  while (e.key < a[i].key)
  { a[i+1] = a[i];
    i--;
  }
  a[i+1] = e;
}
```

program: Insertion into a sorted list.

void insertionSort(element a[], int n)

```
{ int j;
  for (j = 2; j <= n; j++)
  { element temp = a[j];
    insert(temp, a, j-1);
  }
}
```

program: Insertion sort.

### III. Quick sort

\* In quicksort, we select a pivot record from among the records to be sorted.

\* Now, the sorted records are reordered so that the keys of records to the



④

left of the pivot are less than or equal to that of the pivot,

\* records to the right of the pivot are greater than or equal to that of the pivot.

\* finally the records to the left of the pivot and those to its right are sorted independently.

program: Quick Sort

```
void quicksort (element a[], int left,  
                int right)
```

```
{  
    int pivot, i, j;  
    element temp;  
    if (left < right)
```

```
{  
    i = left ; j = right + 1;  
    pivot = a[left].key;
```

```
    do {
```

```
        do i++; while (a[i].key < pivot);
```

```
        do j--; while (a[j].key > pivot);
```

```
        if (i < j) SWAP (a[i], a[j], temp);
```

```
    }
```

```
    while (i < j);
```

```
    SWAP (a[left], a[j], temp);
```

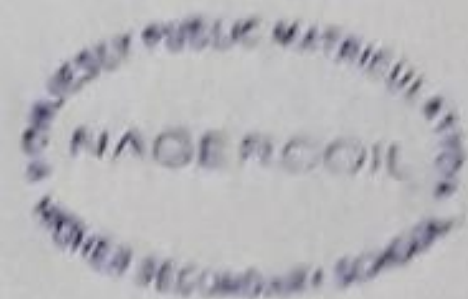
```
    quicksort (a, left, j-1);
```

```
    quicksort (a, j+1, right);
```

```
} }
```



## Mergesort: - IV.

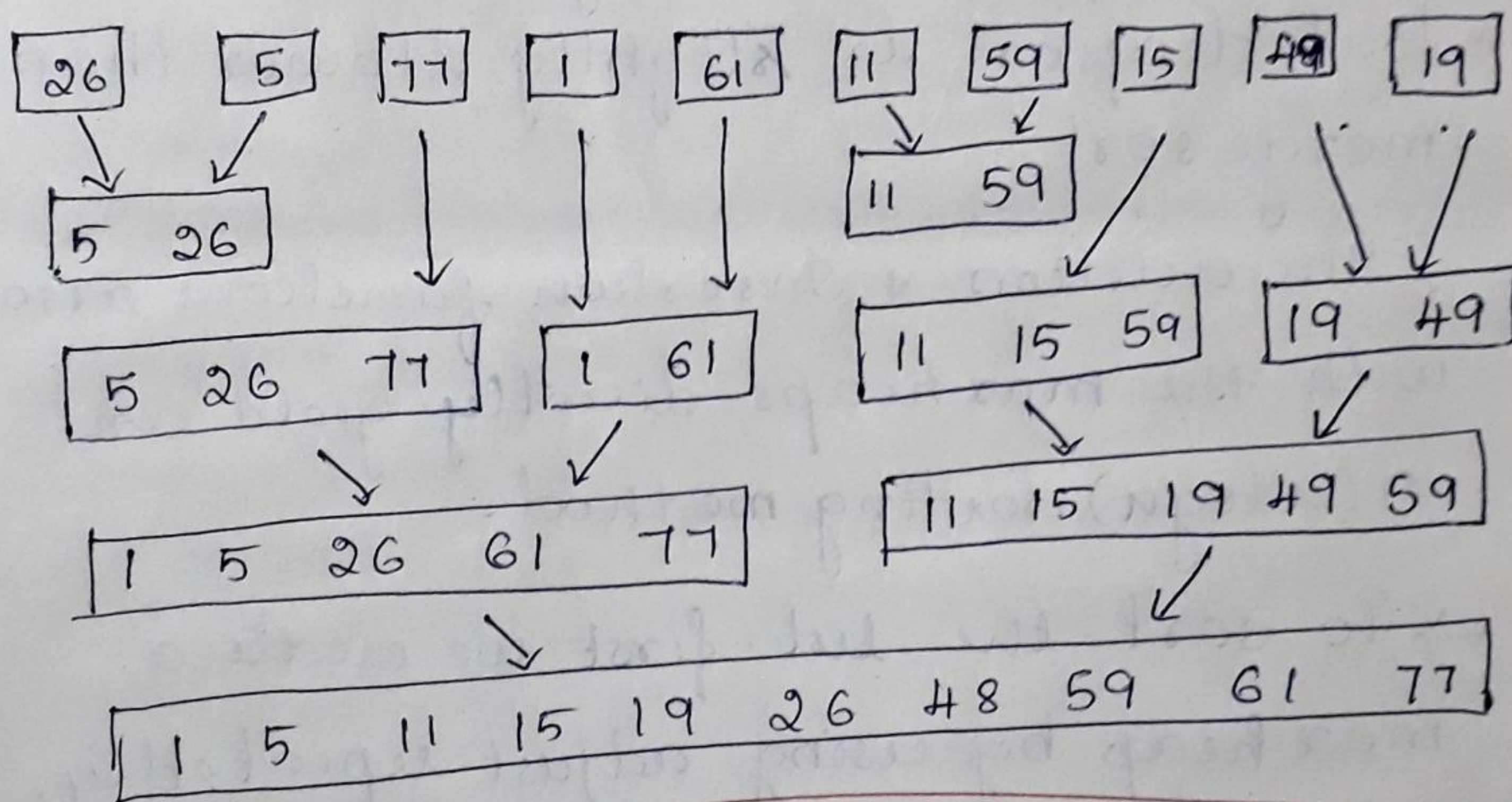


5

### Recursive merge sort: -

In the recursive formulation, we divide the list to be sorted into two roughly equal parts called the left and right sublists. These sublists are sorted recursively, and the sorted sublists are merged.

eg: Input list (26, 5, 77, 1, 61, 11, 59, 15, 49, 19)



program: Recursive merge sort

```
int mergesort(element a[], int left [],  
               int left, int right)
```

```
{  
    if (left >= right) return left;  
    int mid = (left + right) / 2;
```



```

return listMerge (a, link,
                  rmergesort(a, link, left, mid),
                  rmergesort(a, link, mid+1, right));
}

```

#### IV Heapsort: -

\* The Heapsort requires only a fixed amount of additional storage and at the same time as its worst case and average computing time  $O(n \log n)$ . ~~However~~

\* But Heapsort is slightly slower than the mergesort.

\* The deletion & Insertion functions associated with the max heaps directly yield an  $O(n \log n)$  sorting method.

→ To sort the list, first we create a max heap by using adjust repeatedly,

\* Next, we swap the first and last records in the heap.

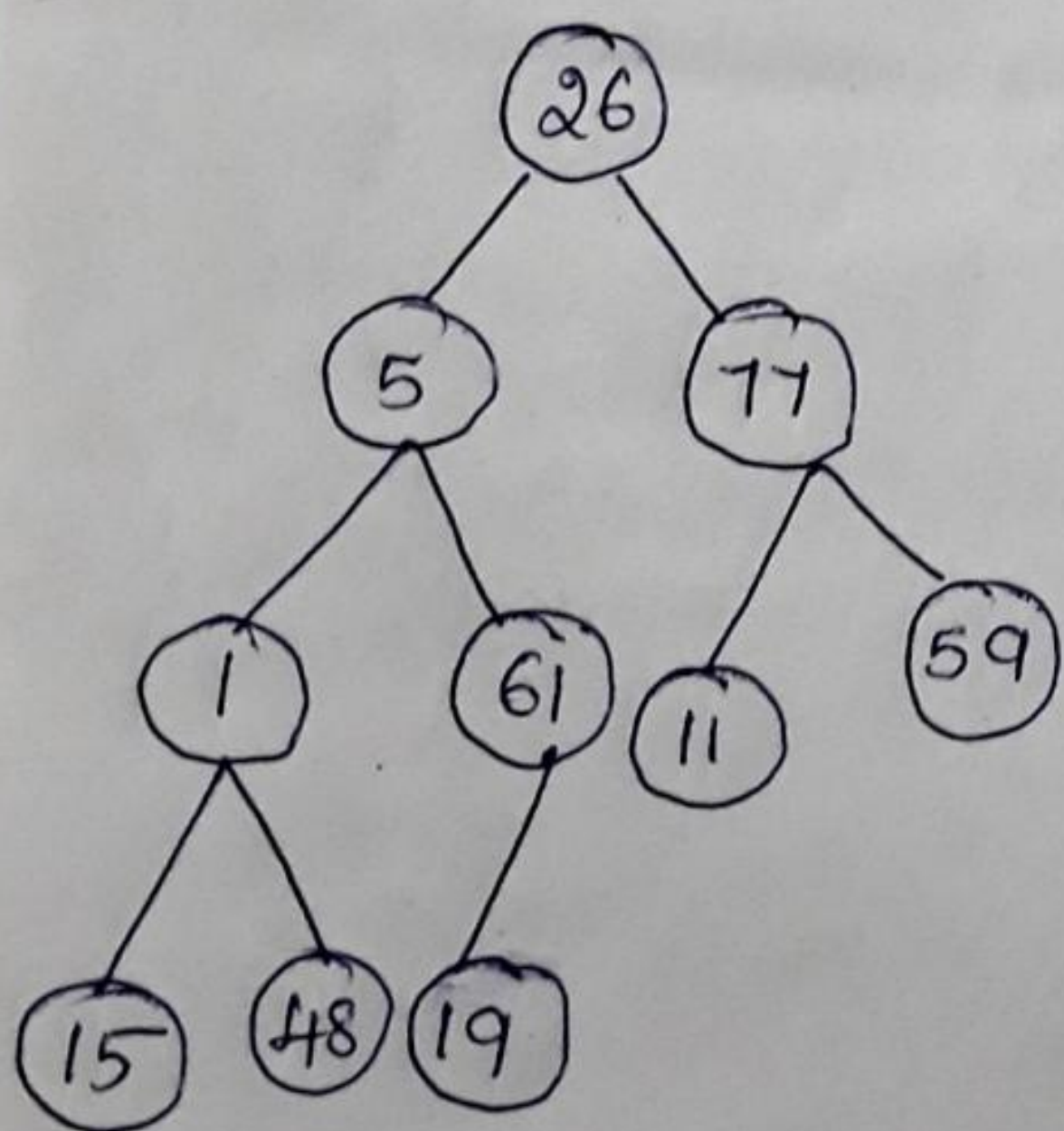
\* since the first record has the maximum key, the swap moves the record with maximum key into its correct position in sorted array.



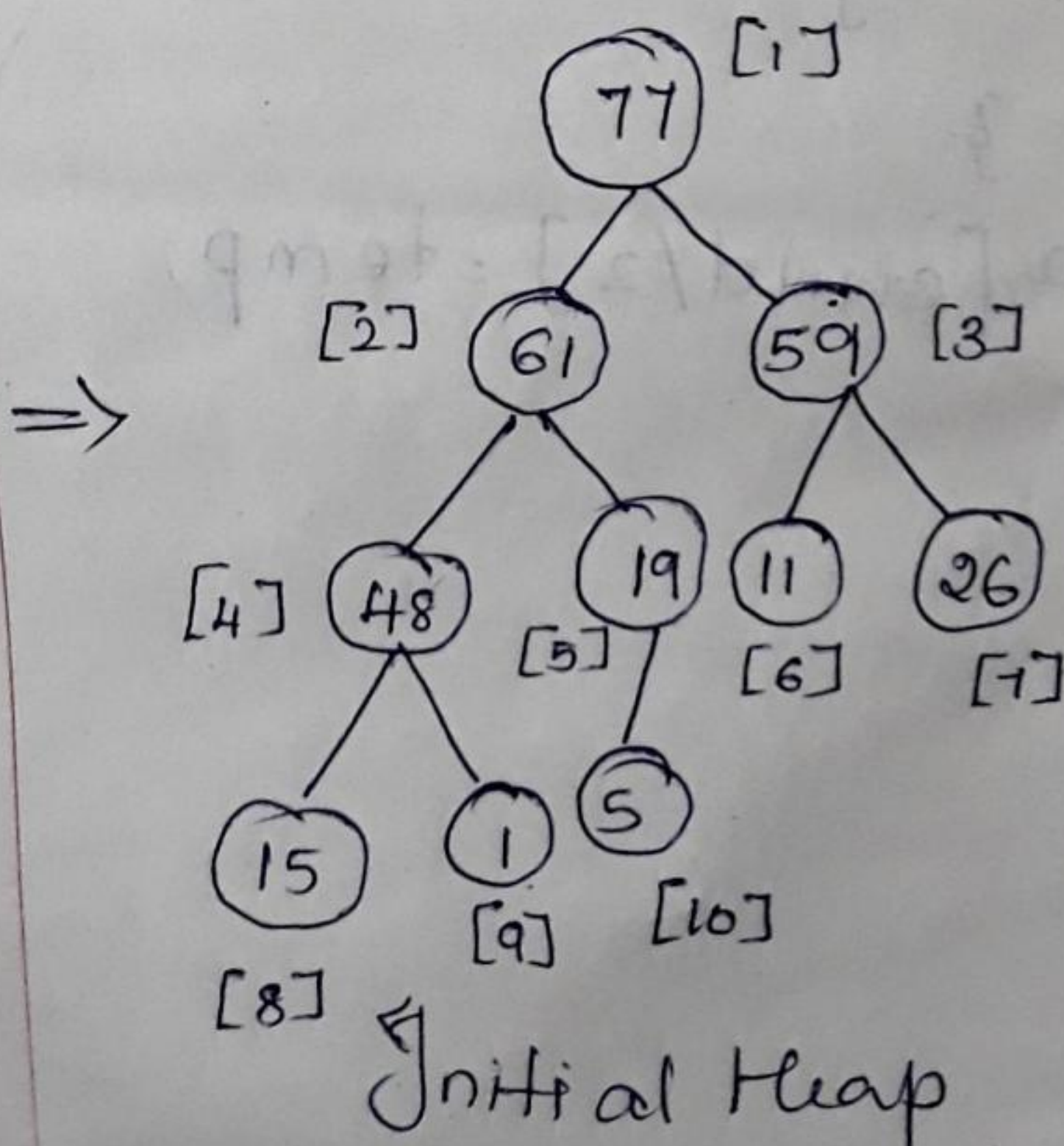
→ Then decrement the heap size and readjust the heap

→ heap process repeated  $n-1$  times to sort the entire array. Each repetition process is called pass.

```
void heapsort (element a[], int n)
{
    int i, j;
    element temp;
    for (i = n/2; i > 0; i--)
        adjust (a, i, n);
    for (i = n-1; i > 0; i--)
    {
        swap (a[i], a[1], temp);
        adjust (a, 1, i);
    }
}
```

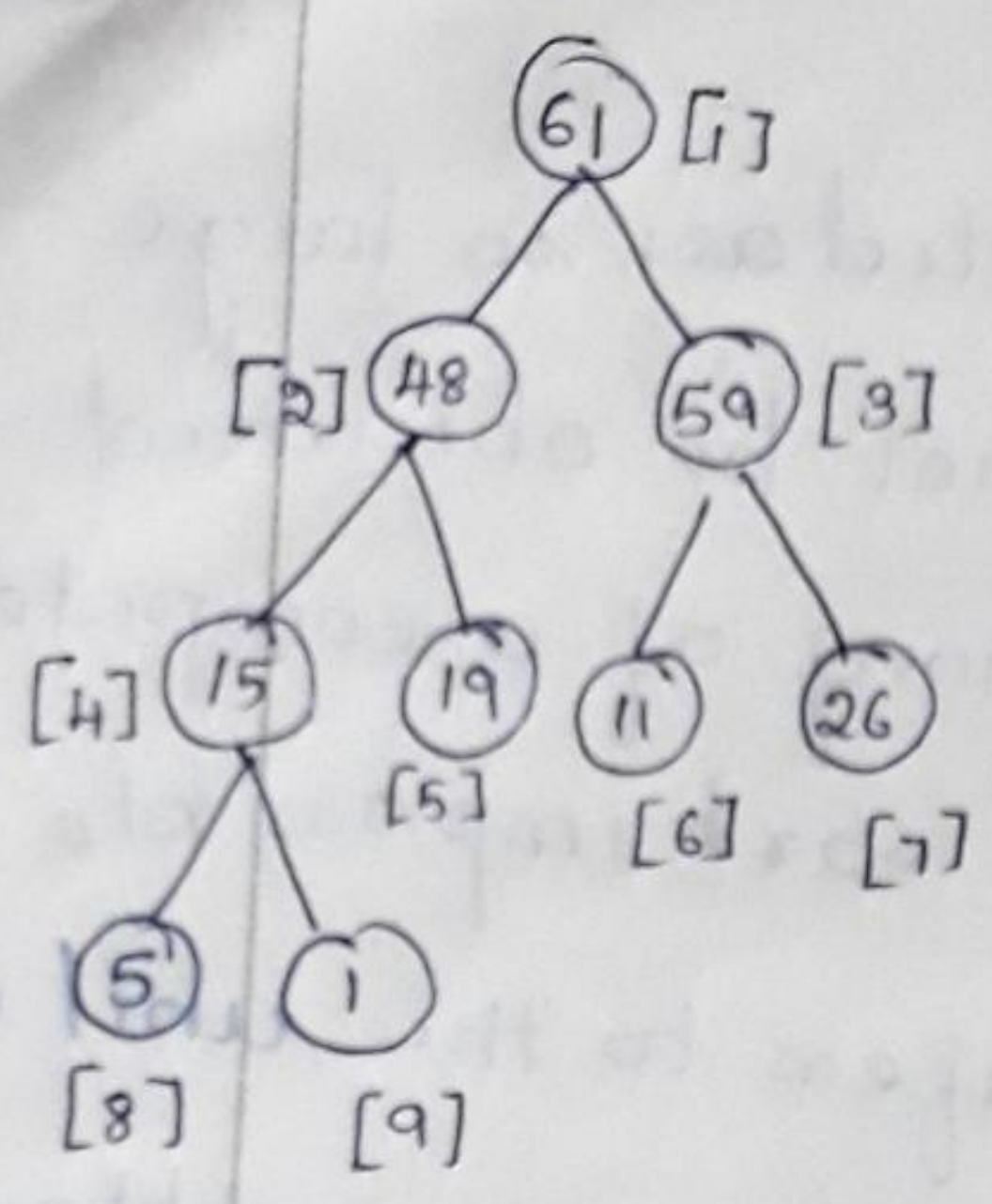


Input array

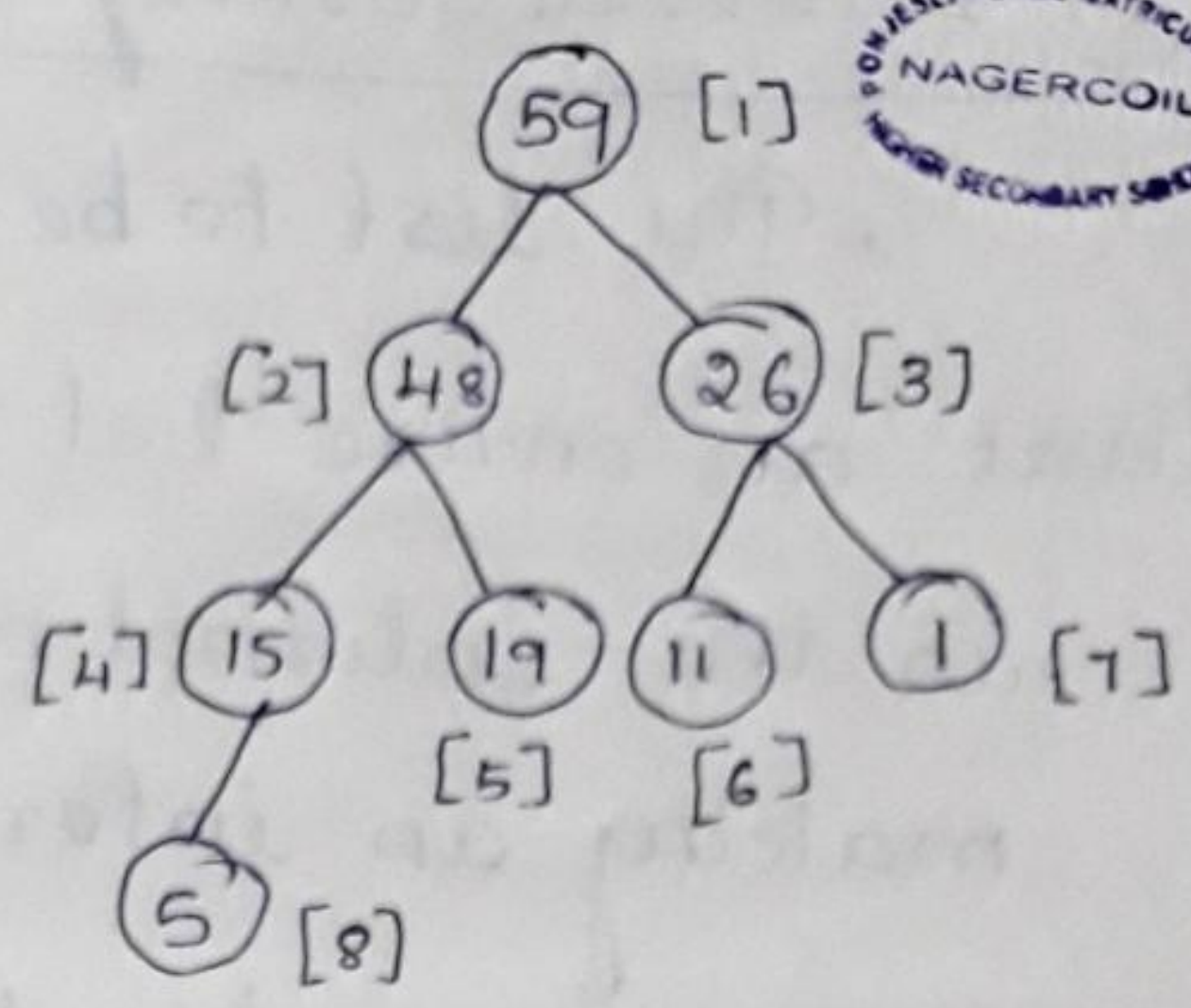


Initial Heap

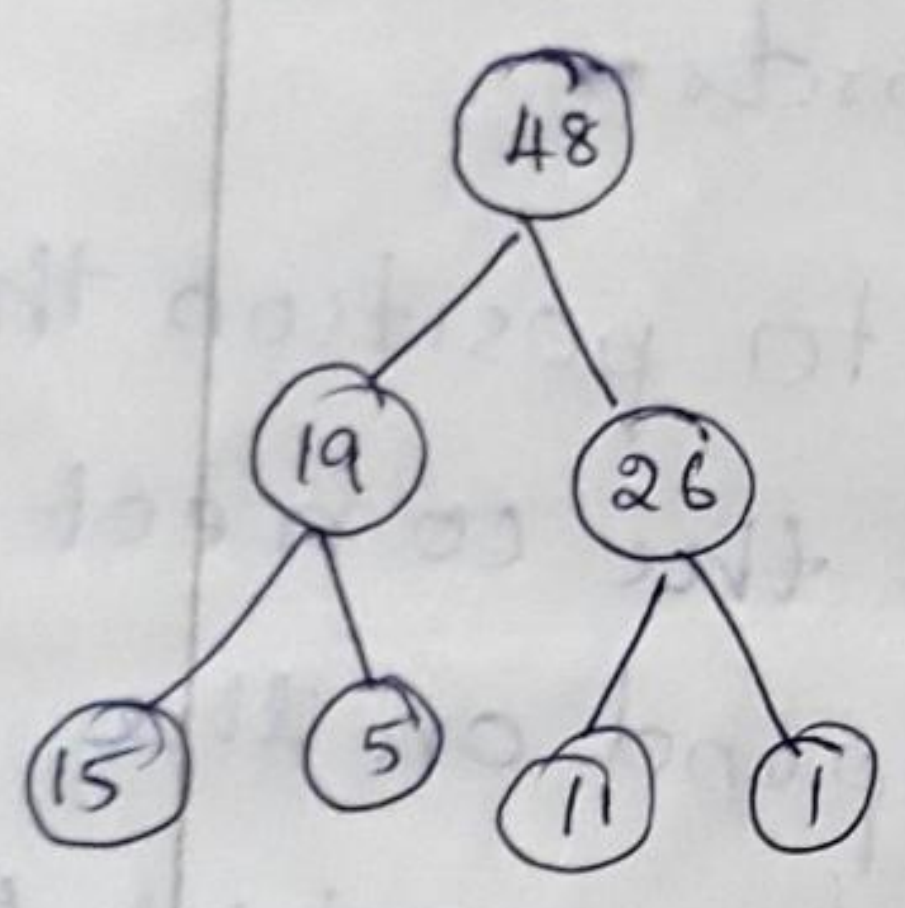




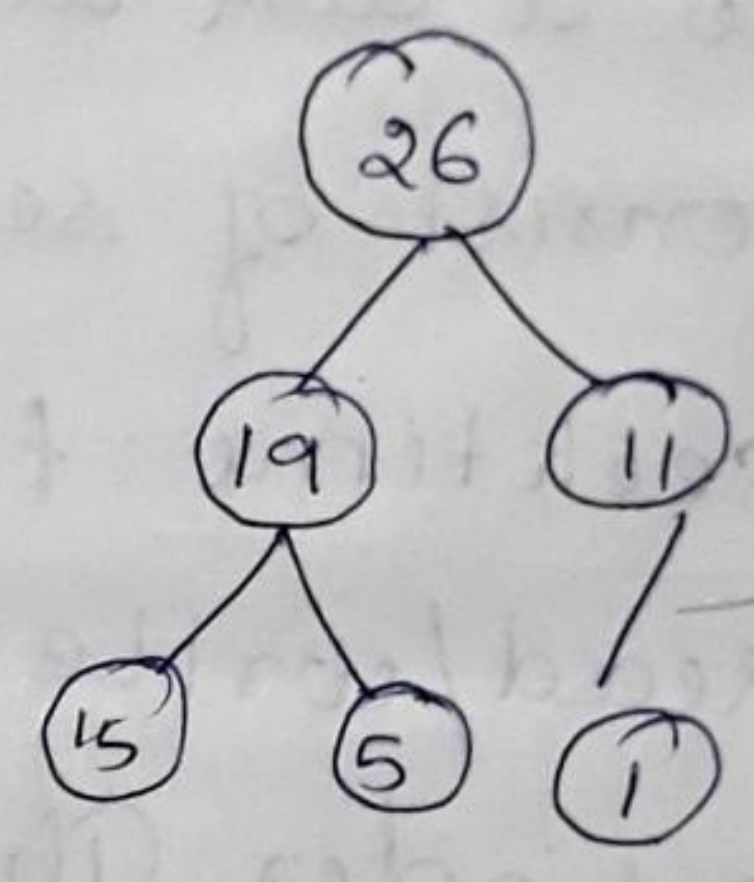
Heap size = 9  
Sorted [77]



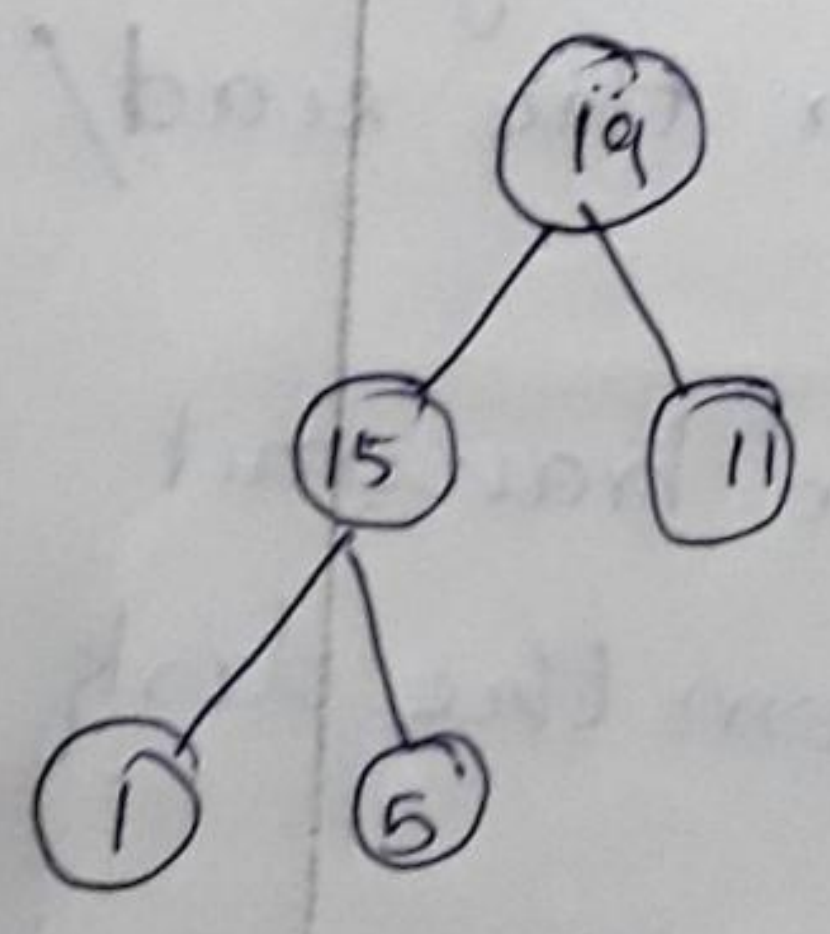
Heap size = 8  
Sorted = [61, 77]



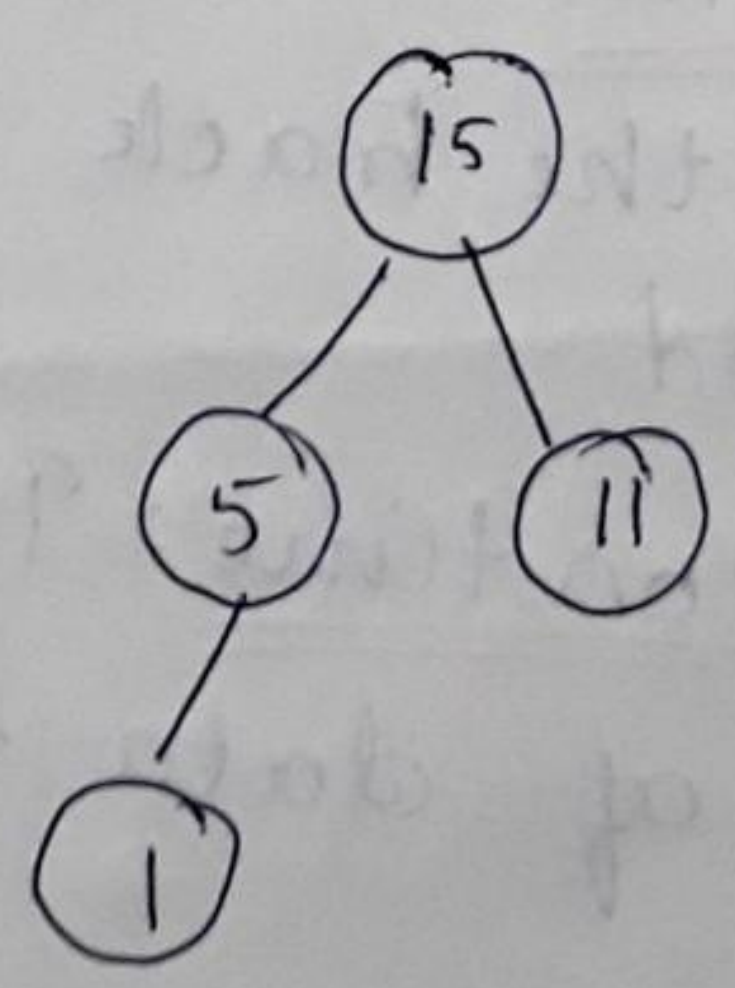
Heap size = 7  
Sorted = [59, 61, 77]



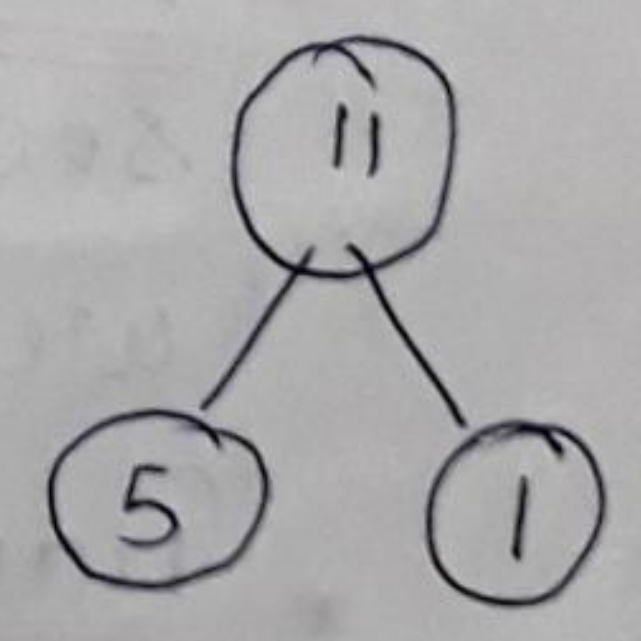
Heap size = 6  
Sorted = [48, 59, 61, 77]



Heap size = 5  
[6, 48, 59, 61, 77]



Heap size = 4  
[19, 26, 48, 59, 61, 77]



Heap size = 3  
[15, 19, 26, 48, 59, 61, 77]



## External sorting

(9)

\* The list to be sorted are so large that an entire list cannot be obtained in the internal memory of a computer, making an internal sort impossible.

→ The term block refers to the unit of data that is read from or written to a disk at one time. block generally consist of several records.

\* Seek time: time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.


\* Latency time: time until the right sector of the track under the read/write head

\* Transmission time: Time to transmit the block of data to/from the disk.

\* The most popular method for sorting on external storage device is merge sort.



\* This method consist of two distinct phases.

(10) 

① first, segments of the input list are sorted using a good internal sort method sorted segments known as runs.

② second, the runs generated in phase one are ~~onto external storage as they are generated~~ merged together to form merge tree.

$t_s$  = maximum seek time

$t_l$  = maximum latency time

$t_{rw}$  = time to read or write one block of 250 records

$t_{io}$  = time to input or output one block

$$t_{io} = t_s + t_l + t_{rw}$$

$t_{is}$  = time to internally sort 150 records

$n t_m$  = time to merge  $n$  records from input buffers to the output buffer.



## a) k-way merging:-

\* The two-way merge function merge is almost identical to the merge function.

\* The number of passes over the data can be reduced by using higher order merge and simultaneously merge  $k$  runs together.

\* Input/output time may be reduced by using higher-order merge.

\* Total number of key comparison is

$$n(k-1)\log_k m = n(k-1)\log_2 m / \log_2 k$$

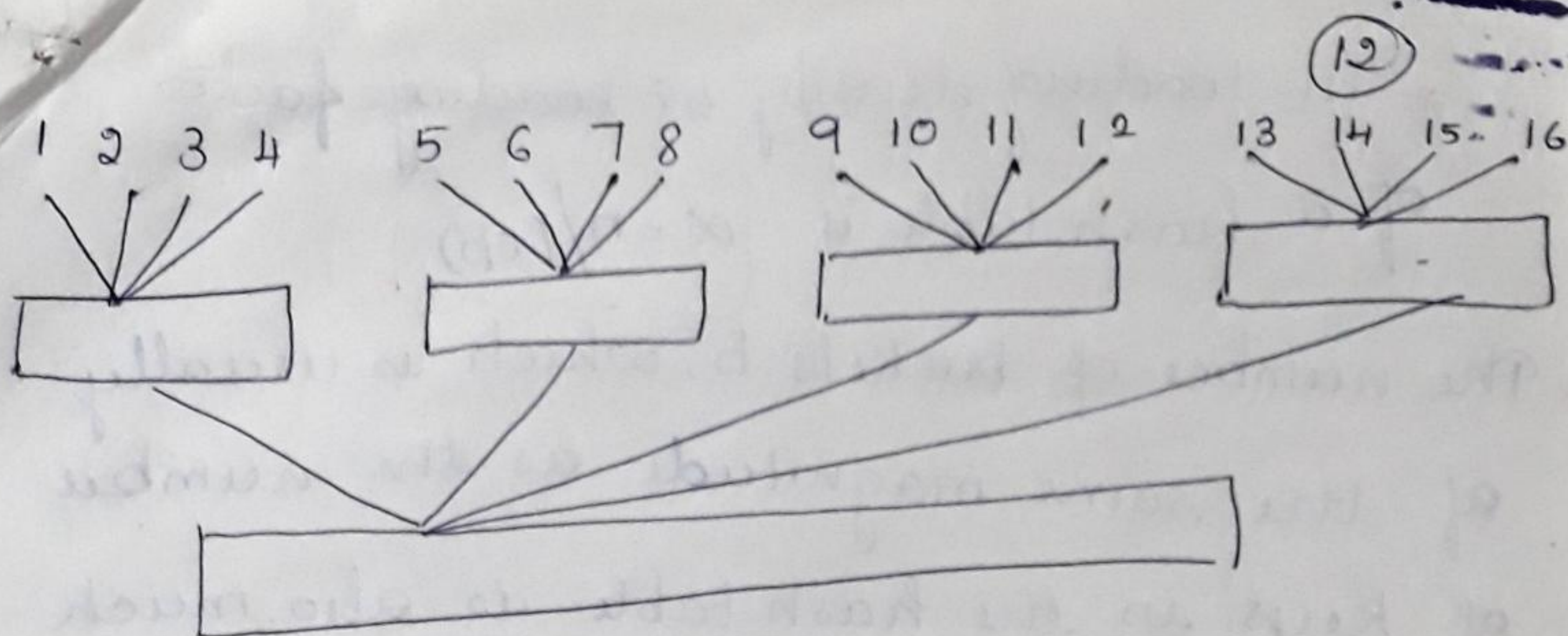
$n \rightarrow$  number of records in the list.

$(k-1)/\log_2 k$  factor by which the number of key comparison increases

If  $k$  increases the reduction in input/output time will be outweighed

by the resulting increase in CPU time needed to perform  $k$ -way merge.





four way merge on 16 runs

## VIII Static Hashing

### a) Hash tables:

In static Hashing, the dictionary pairs are stored in a table  $ht$ , called the hash table. The hash table is partitioned into  $b$  buckets,  $ht[0], \dots, ht[b-1]$ .

\* Each bucket is capable of holding  $s$  dictionary pairs.

\* bucket consists of slot, each slot being large enough to hold one dictionary pair.

\* The key density of a hash table is the ratio  $\boxed{n/T}$ , where  $n$  is the number of pairs in the table &  $T$  is the total number of possible keys.



\* The loading density or loading factor of a hash table is  $\alpha = n/(sb)$

\* The number of buckets  $b$ , which is usually of the same magnitude as the number of keys, in the hash table is also much less than  $T$ . Therefore the hashfunction  $h$  maps several different keys into the same bucket.

\* Two keys  $k_1$  &  $k_2$  are said to be synonymous with respect to  $h$  if  $h(k_1) = h(k_2)$ .