Unit 1

Data Structures

Programming methodology deals with different methods of designing programs.

- **Programming Methoddogy** : The approach of analysing complex problems, planning for software development and controlling the development process is called programming methoddogy

Types –

(i) **Procedural Programming** ~ Solving a problem by breaking it down into procedures or blocks.

(ii) **Object Oriented Programming** ~ Here, the solution revolves around objects that are a part of the problem.

(iii) **Functional Programming** ~ Problem is broken down into 'functional' units. Then units are stitched to form complete solution.

(iv) **Logical Programming** ~ Here, the program is broken down into logical units.

- **Design of Algorithms**

- **Abstract Data Types** : ADT is a type (or class) for objects whose behaviour is defined by a set of value or a set of operations.

The definition of ADT only mentions what operations are to be performed but not how they will be performed. 3 types are –

(i) **List ADT** – Elements arranged in sequential order.

get() Return the element from list
insert()
remove()
removeAt()
size()
isEmpty()
isFull()

(ii) **Stack ADT** – Elements of same type arranged in sequential order. LIFO

push()
pop()
peak() ~ Return the element at the top of stack without removing
size()
isEmpty()
isFull()

(iii) **Queue ADT** – FIFO Addition at end & deletion from front.

enqueue() Insert
dequeue() Remove
peak()
size()
isEmpty()
isFull()

ADT can be implemented using Arrays, singly linked list, doubly linked list etc.

Page No.
Date

Algorithm : Step - by - Step finite sequence of instruction to solve well defined computational problem.

* **Steps for creating a programs**

Mathematical Model / Informal Algorithm	⇒	Formal Language / Pseudo-Code/ Program	⇒	Finally write a program in some language.

* **Primitive Data type** — Integer, Float, Character

Non - primitive data structure — Array, List, Files

- Array → Linear
- List → Linear: Stack, Queue; Non Linear: Graph, Trees

* **Abstract Data Type** : It is logical description of how we views the data & operations that are allowed without regard to how they will be implemented.

* **1D Array Address Calculation**

$A[i] = B.A. + i \times \text{Datatype}$

$A[k] = \text{Address of } A[LB] + (k - LB) * \text{Size of Datatype}$

Page No.
Date

* **Row Major** (row col)

$A[m][n]$

Address

$A[i][j] = B.A + (i \times n + j) \times \text{Size of Datatype}$

* **Column Major** (row col)

$A[m][n]$

Address

$A[i][j] = BA + (j \times m + i) \times \text{Size of Datatype}$

* **N - Dimentional Array Address Calculation**

$A(m_1, m_2, m_3, \ldots\ldots m_n)$

Find address of $A[k_1][k_2][k_3]\ldots[k_n]$ where $1 \le k_1 \le m_1$, $\ldots$ $1 \le k_n \le m_n$.

$L_i = UB - LB + 1$ & $E_i = k_i - LB$ ✓

Row Major Order (Last subscript very first)

$A[k_1, k_2, \ldots k_n] = BA + w \times [(\ldots((E_1 L_2 + E_2) L_3 + E_3) L_4 + \ldots E_{N-1}) L_N + E_N)]$

(w = Size of Datatype)

$E \rightarrow$ Effective Address $L \rightarrow$ Length

Eg. $M(2:8, -4:1, 6:10)$

Find $M[5, -1, 8]$.

Ans. $BA = 200$; $L_1 = 8 - 2 + 1 = 7$; $L_2 = 1 - (-4) + 1 = 6$

$L_3 = 10 - 6 + 1 = 5$; $E_1 = 5 - 2 = 3$ $E_2 = -1 - (-4) = 3$

$E_3 = 8 - 6 = 2$

$E_1 L_2 = 18$; $E_1 L_2 + E_2 = 21$; $(E_1 L_2 + E_2) L_3 = 105$

$(E_1 L_2 + E_2) L_3 + E_3 = 107$

Page No.
Date

$w = 4$ (Way)

$M[5, -1, 8] = 200 + 4 \times 107 = 200 + 428$
$= 628$

* Column Major Order – N-Dimensional Address Calculation

$A[k_1, k_2, k_3 \ldots k_N] = BA + w \times ((((\ldots E_N L_{N-1} + E_{N-1}) L_{N-2} \ldots \ldots E_2) L_1 + E_1) \ldots))$

Eg. $B[1:8, -5:5, -10:5]$

find $B[3][3][3] = ?$

$L_1 = 8 - 1 + 1 = 8 \quad L_2 = 5 + 5 + 1 = 11 \quad L_3 = 5 + 10 + 1 = 16$
$E_1 = 3 - 1 = 2 \quad E_2 = 3 - (-5) = 8 \quad E_3 = 3 - (-10) = 13$

$E_3 L_{3-1} = E_3 L_2 = 13 \times 11 = 143$
$E_3 \times L_2 + E_2 = 143 + 8 = 151$
$(E_3 L_2 + E_2) L_1 = 151 \times 8 = 1208$
$(E_3 L_1 + E_2) L_1 + E_1 = 1208 + 2 = 1210$

$\therefore A[3][3][3] = 4000 + 4 \times 1210 = 5240$ (Csay)

Page No.
Date

* Array Operations

```
void main()
{
  int ch;
  do{
      printf("Enter the operation you wanna perform:");
      printf("1. Create 2. Display 3. Insert 4. Delete
              5. Search 6. Sort 7. Merge 8. Exit");
      scanf("%d", &ch);
      switch(ch)
      {
        case 1: create();
                break;
        case 2: display();
                break;
        case 3: insert();
                break;
        case 4: delete();
                break;
        case 5: search();
                break;
        case 6: sort();
                break;
        case 7: merge();
                break;
        case 8: exit(0);
                break;
        default:
                printf("Invalid choice! Press any key to exit.");
                break;
      }
```

```
void create()                    // Creating an array
{
  printf("Enter the size of array: ");
  scanf("%d", &n); printf("Enter the elements: ");
  for(i=0; i<n; i++)
    {
      scanf("%d", &a[i]);
    }
}
void display()                   // Displaying the array
{
  int i;
  printf("The array elements are: ");
  for(i=0; i<n; i++)
  {
    printf("%d", &a[i]);
  }
}
void insert()                    // Inserting element at position
{
  printf("Enter the position to insert new element:");
  scanf("%d", &pos);
  printf("Enter the value of new element:");
  scanf("%d", &val);
  for(i=n-1; i>=pos; i--);
  {
    a[i+1] = a[i]
  }
  a[pos] = val;
  n = n+1;
}
```

```
void delete()                    // Deleting element from position
{
  printf("Enter the position of the element to be deleted:");
  scanf("%d", &pos); val = a[pos];
  for(i=pos; i<n-1; i++)
    {
      a[i] = a[i+1];
    }
    n = n-1;
    printf("The deleted element is:%d", &val);
}
void search()                    // Linear Search
{
  printf("Enter the element to be searched:");
  scanf("%d", &key);
  for(i=0; i<n; i++)
    {
      if(a[i]== key)
        {
          printf("Element found at position:%d", i);
    }
}
void sort()
{
  for(i=0; i<n-1; i++)
    {
      for(j=0; j<n-1-i; j++){
      if(a[j] > a[j+1])
        {
          temp = a[j];
          a[j] = a[j+1];
          a[j+1] = temp; }
    }
}
```

```
    printf("Array after sorting is ~%d", );
    display();
}
```

* **Algorithm**

Array Traversal

Step 1 Repeat for $i = LB$ to UB (i = i+1)
2 Process A[i]
[END of Loop]
3. END

Array Deletion

1. Set item = A[loc]
2. Repeat for i = pos to N
3. Set A[i] = A[i+1]
[END of LOOP]
4. Set N = N - 1
5. END.

Insertion

1. Set i = n
2. Repeat while (i > loc)
3. Set A[i+1] = A[i]
4. Set i = i - 1
5. [END of loop]
6. A[loc] = item
7. n = n + 1
8. END

- **Stack ADT**: A list with the restriction that insertion & deletion can be performed from only one end, i.e. top.

* In ADT, we only talk about operation and not implementation, i.e. push, pop, isEmpty, size, etc.

Stack using Array

Push(x)	pop(x)	top()	isEmpty

```
push(x)
{
    top = top+1
    A[top] = x
}

pop()
{
    top = top-1
}

top()
{
    return A[top]
}

isEmpty()
{
    if(top == -1)
        return True
}
```

- **Infix, Prefix, Postfix Notation**

Prefix	Infix	Postfix
/*+ABCD	(A+B)*C/D	AB+C*D/

Algorithm for Postfix

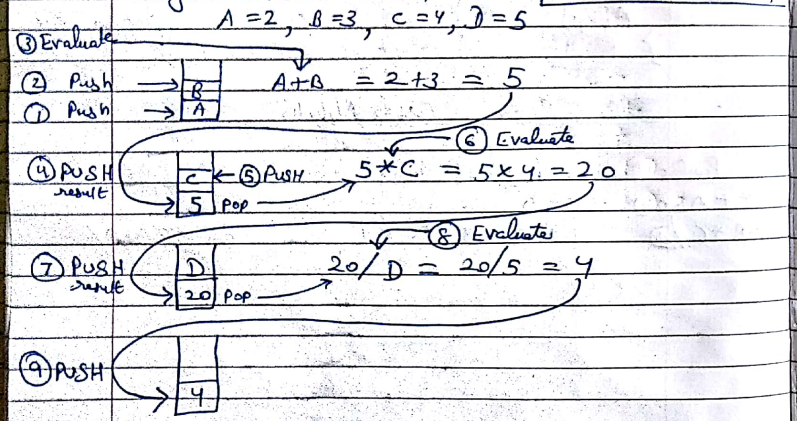
Let R be a postfix expression.

1. Scan R from left to right & repeat step 2 & 3 for each element until stack is empty.
2. If an operand is encountered, put it on stack.
3. If an operator is encountered
   (a) Pop two elements from stack.
   (b) Evaluate the expression formed by two operands & the operator.
   (c) PUSH result of (b) on stack.
4. SET result equal top element of stack.
5. END

LEFT TO RIGHT SCANNING

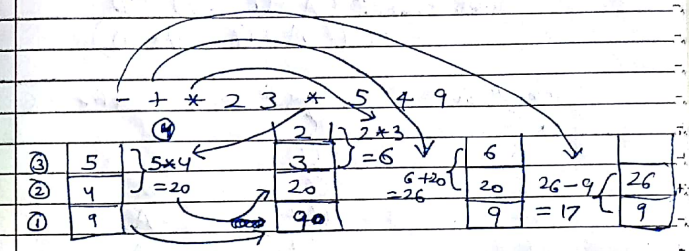
E.g. AB + C * D /

A = 2, B = 3, C = 4, D = 5



* Evaluation of Prefix

1. Right to left scanning.
2. If operand is there, PUSH
3. If operator is there, POP; last two elements.



[(2*3)+(5*4)] - 9 = 17

* Conversion Using Stack

Infix to Postfix

Let S be an arithmetic expression written in infix notation & this algo will convert S into postfix notation R. Steps –

1. PUSH parenthesis to end of S.
2. Scan S from left to right until stack is empty. Repeat step 3 to 6.
3. If an operand is encountered, ADD to R.
4. If a left parenthesis is encountered, push in stack
5. If an operator is encountered,
   (a) Add operator on the stack.
   (b) If there is another operator on top of stack & new operator has less or equal precedence, then pop stack & add to R. And push new operator on Stack.

6. If right parenthesis is encountered,
(a) Pop from Stack & add to R.
(b) Remove left parenthesis.
7. END.

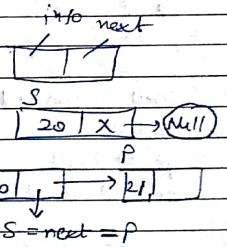
E.g. S : (A+(B-C)) *D
⇒ ((A+(B-C))* D)

Parenthesis	Stack	Postfix - R
(	(	
(	((	
A	((	A
+	((+	A
(	((+(	A
B	((+(	AB
-	((+(-	AB
C	((+(-	ABC
)	((+	ABC-
)	(	ABC-+
*	(*	ABC-+
D	(*	ABC-+D
)		ABC-+D*

* LINKED LIST : It is a dynamic data structure. They can grow & shrink during execution of program.

→ Efficient
→ Linear (according to access strategy)
→ Non-linear (according to storage)

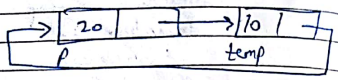
* Single Linked List



STRUCTURE

```
#include<stdio.h>
#include<alloc.h>
Struct node
{
    int info;
    node * next;
};
main()
{
Struct node *S;
S = (struct node*) malloc (sizeof(struct node));
   Type casting          // Dynamic Memory allocation
S → info = 20;
S → next = NULL;
printf("%d", S → info);
}
```

* Circular Linked List →



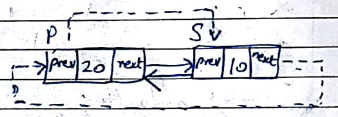

```
struct node
{
  int info;
  node *next;
};

struct node *s, *p, *temp;
p = (struct node*) malloc(sizeof(struct node));
p → info = 20;

temp = (struct node*) malloc(sizeof(struct node));
temp → info = 10;
p → next = temp;
temp → next = p;
```

Circular Linked List is a linked list where all nodes are connected to form a circle. There is no null pointer in circular linked list.

* Doubly Linked List




```
struct node
{
  int info;
  node *next, *prev;
};

struct node *s, *p
P = (struct node*) malloc(sizeof(struct node));
P → info = 20;
S = (struct node*) malloc(sizeof(struct node));
S → info = 10;
```

Doubly

```
P → next = S;      P → prev = NULL;
S → prev = P;      S → next = NULL;
```

OR

```
+ S → next = P;
+ P → prev = S;
```

* Queue ADT ! It is linear data structure & works on FIFO

* Queue ADT : Operation only

A list or collection with restriction that insertion can be performed at one end (Rear) & deletion can be performed at other end (Front).

void

(i) enqueue(x) → Insert(x)
(ii) dequeue() → deletion from front.
(iii) front()
(iv) isEmpty() → T/F

* Circular Queue : Insertion of new element is done at every first location of queue is full.

Insertion → Rear = (Rear + 1) % MaxSize

Page No.
Date

Algo for insertion

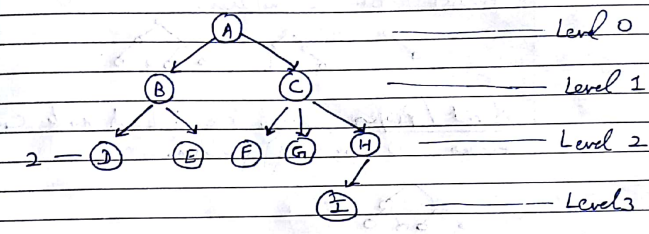
```
if (front == (rear+1) % MaxSize)
    printf ("Queue Overflow !! ");
else if (front == -1)
    set front = rear = 0;
else rear = (rear+1) % MaxSize;
Queue[Rear] = value;
END of if loop
EXIT
```

Algo for Deletion

```
if (front == -1)
    print (Underflow)
else (item = queue[front])
if (front == rear)
    set front = -1
    rear = -1
else (front = front+1) % MaxSize;
END of IF
EXIT
```

Page No.
Date

Trees

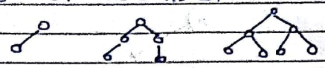


- Collection of entity are nodes.
- A is the root node.
- B, C are children of A.
- A node not having any child is called leaf node.
- D, E, F, I are leaf nodes.
- A is grandparent of D.
- D & E are descendant of A.
- E & F are cousins.
- D & E are siblings.

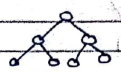
* Tree is a recursive data structure.
* Application of trees :-
  - Windows file system.
  - Linux file system
  - Trie → Dictionary
  - Networking, cloud computing
* Tree with N nodes have N-1 edges.
* Depth of x := length of path from root to x.
  = No. of edges in path from root to x.
* Height of tree : No. of edges in longest path from x to leaf.
  Height of tree = height of root node.

Page No.
Date

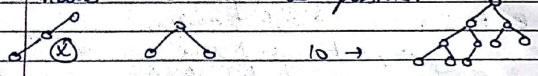
* **Binary Tree** : Each node can have at most of 2 children.



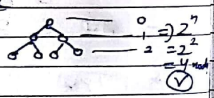
* **Strict / Proper Binary Tree** : Each node can have either 0 or 2 children.



* **Complete Binary Tree** : All levels except possible the last are completely filled & all nodes are left as possible.



* **Perfect or Fully Complete Binary Tree**



→ Level : has 2^i nodes.

→ Leaves are at level h & no. of leaves $= 2^h$

→ Total internal nodes $= 2^h - 1$
Total no. of nodes $= 2^{h+1} - 1 = n$
Height $= \log_2$ (No. of leaves)

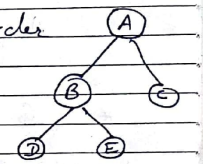
→ No. of internal nodes = no. of leaves $- 1$

Page No.
Date

* **Binary Tree Traversal**

Visiting each node in the tree exactly once in some order.

(i) **Breadth First Traversal** → Level Order



Level 0 → Level 1 → Level 2 → .....

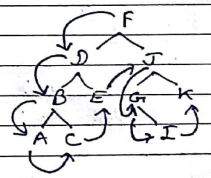
A → B → C → D → E

(ii) **Depth First Traversal**

- **Preorder** → Root, Left, Right : A B D E C
- Inorder → Left, Root, Right : D B E A C
- Postorder → Left, Right, Root : D E B C A

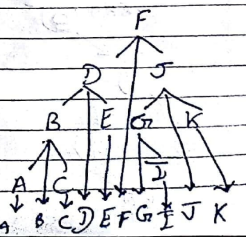
TRICK

Preorder FDBACEJGIK



Inorder

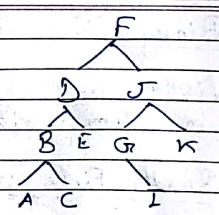
[Freefall]



Page No.
Date

Postorder

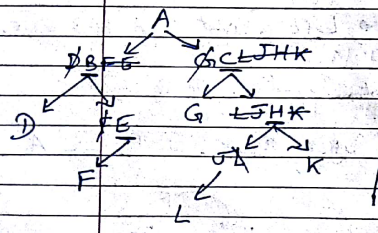
Leaf break from left.



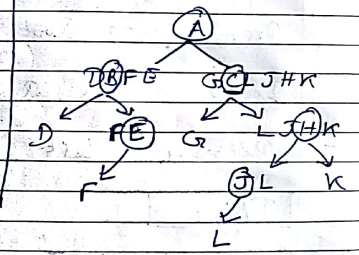
A C B E D I G K J F

* Construct Binary Tree

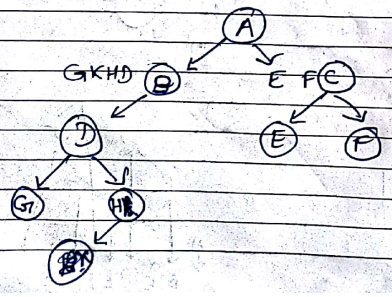
Preorder : A B D E F C G H J L K
Inorder : D B F E A G C L J H K



Postorder : D F E B G L J K H C A
Inorder : D B F E A G C L J H K



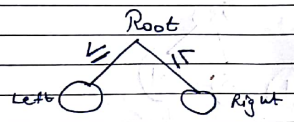
Preorder : A B D G H K C E F
Postorder : G K H D B E F C A



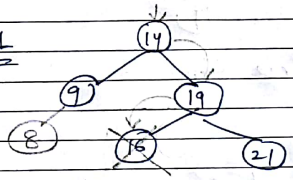
Page No.
Date

* Binary Search Tree (BST)

Complexity → O(log n)

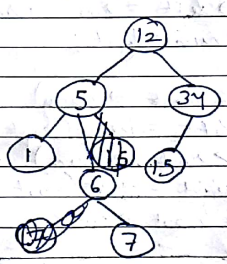


E.g 1

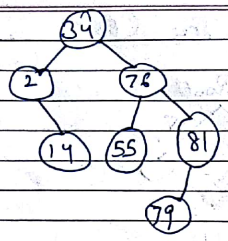


(i) Insert (8)
(ii) Delete (16)

Eg 2 12, 34, 5, 1, 15, 6, 7



Eg 3 34, 76, 2, 14, 81, 55, 79



* Binary Search Tree Algorithm

- Insert a Node

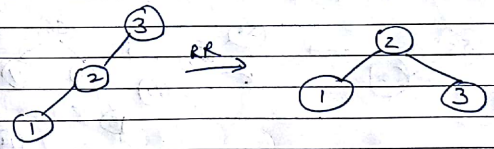
```
node *insert (node *root, int digit)
  {
    if (root == NULL)
      {
        root = (node*) malloc (sizeof (node));
        root → left = root → right = NULL;
        root → num = digit;
      }
  else if (digit < root → num)
      root → left = insert (root → left, digit);
  else if (digit > root → num)
      root → right = insert (root → right, digit);
  else if (digit = root → num)
      {
        printf ("Duplicate value, Couldn't add!");
        exit (0);
      }
  return (root);
}
```

- Search

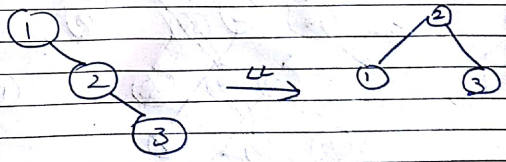
```
void search (node *root, int digit)
{
  if (root == NULL)
    printf ("No. does not exist")
else if (digit == root → num)
    printf ("%d", digit)
else if (digit < root → num)
    search (root → left, digit)
else
    search (root → right, digit)
}
```

* AVL Tree (Height Balanced BST Tree)

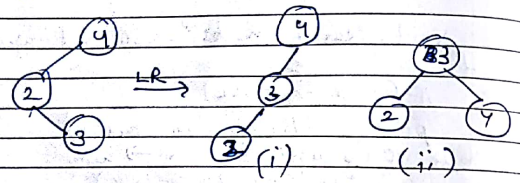
① RR Rotation (RR)



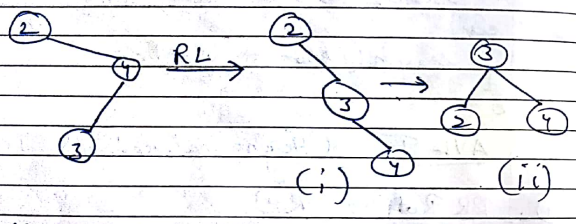
② Left Left Rotation (LL)



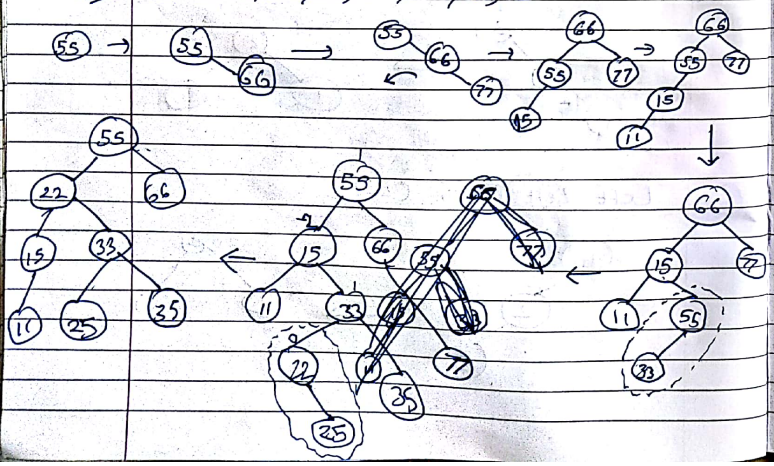
(3) Left Right (LR) Rotation



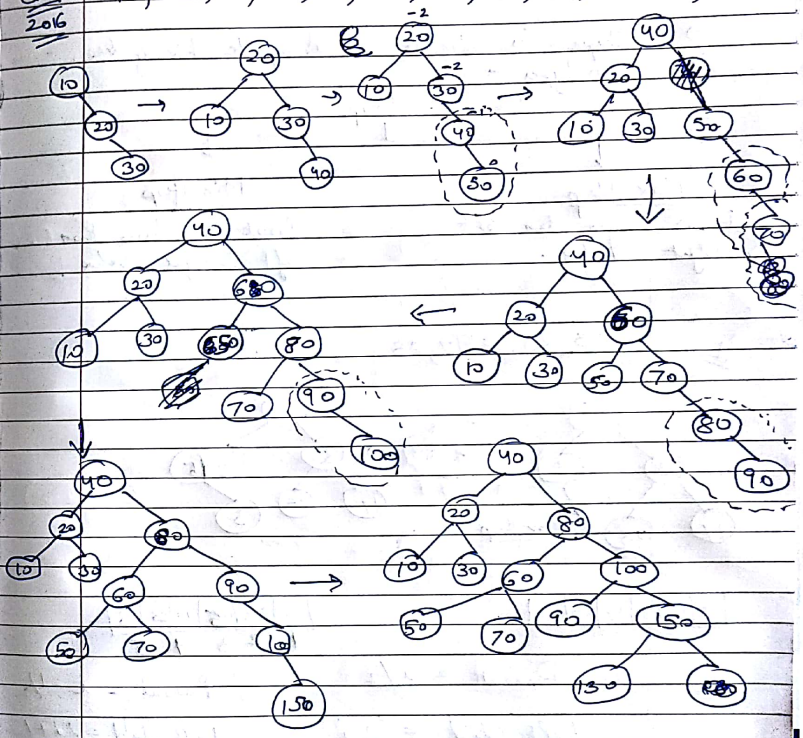
(4) RL Rotation



Eg:1 55, 66, 77, 15, 11, 33, 22, 35, 25



Qs. 2016 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 130, 15



Page No.
Date

Heap

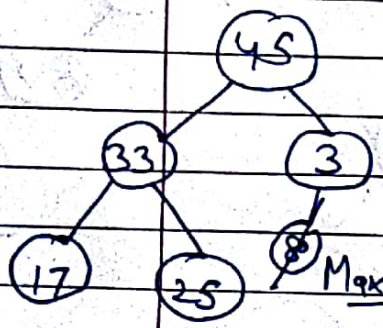
→ Shape is similar to complete binary tree.
→ Two types –

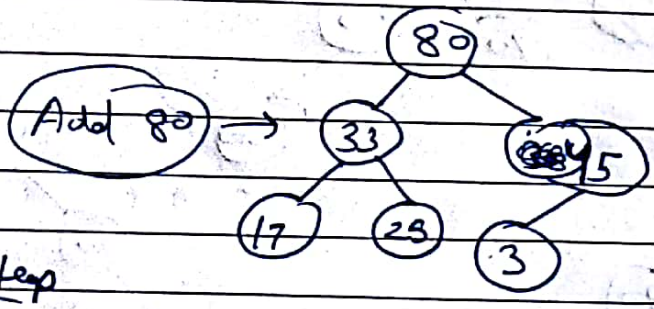
Heap

Max Heap
Root is larger than left & right value.

Min Heap
Root is lesser than left & right value.

Eg. 45, 33, 3, 17, 25





45	33	3	17	25	

80	33	45	17	25	3
1	2	3	4	5	

Parent = $i/2$

Left child = $2i$

Right = $2i+1$

$i=1$

Parent = $i-1/2$

Left child = $2i+1$

Right child = $2i+2$

$i=0$

Page No.
Date

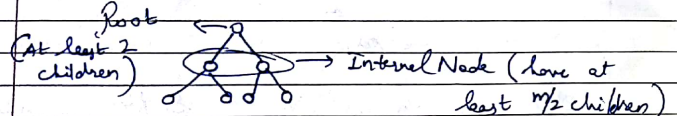
* B-Tree

B-Tree is a balanced M-way tree.

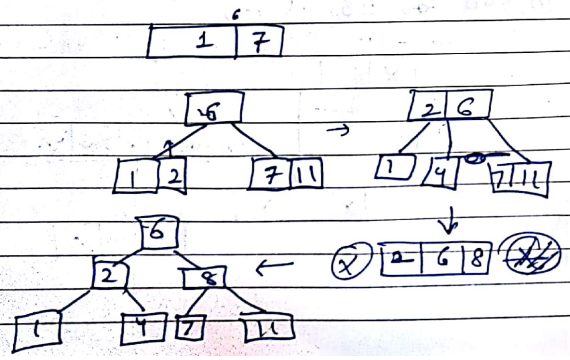
It satisfies the following properties → m+1 children

1. Each node has at most m children.
2. Each internal node has at least $\left\lceil \frac{m}{2} \right\rceil$ children.
3. Root has at least 2 children if it is not leaf.
4. A non leaf node with k children has k-1 keys
5. All leaves appear in same level.

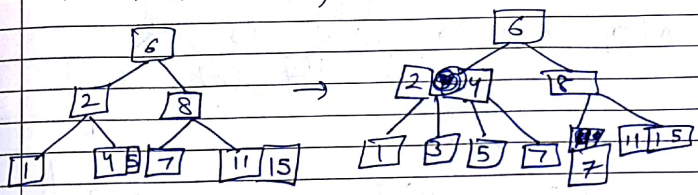
$m = 5 \rightarrow \left\lceil \frac{5}{2} \right\rceil = 2.5 = 3$



Eg B-tree of order 3 → (m=3)
1, 7, 6, 2, 11, 4, 8

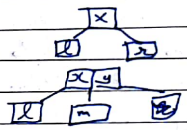


2 1, 7, 6, 2, 11, 4, 8, 5, 15, 3, 12



* 2-3 Tree

- It has 3 different kinds of nodes.
- Leaf Node
- 2-node
- 3-node



* The length of a path from 2 or 3 node to every leaf in its subtree must be same.

* It holds search tree condition.

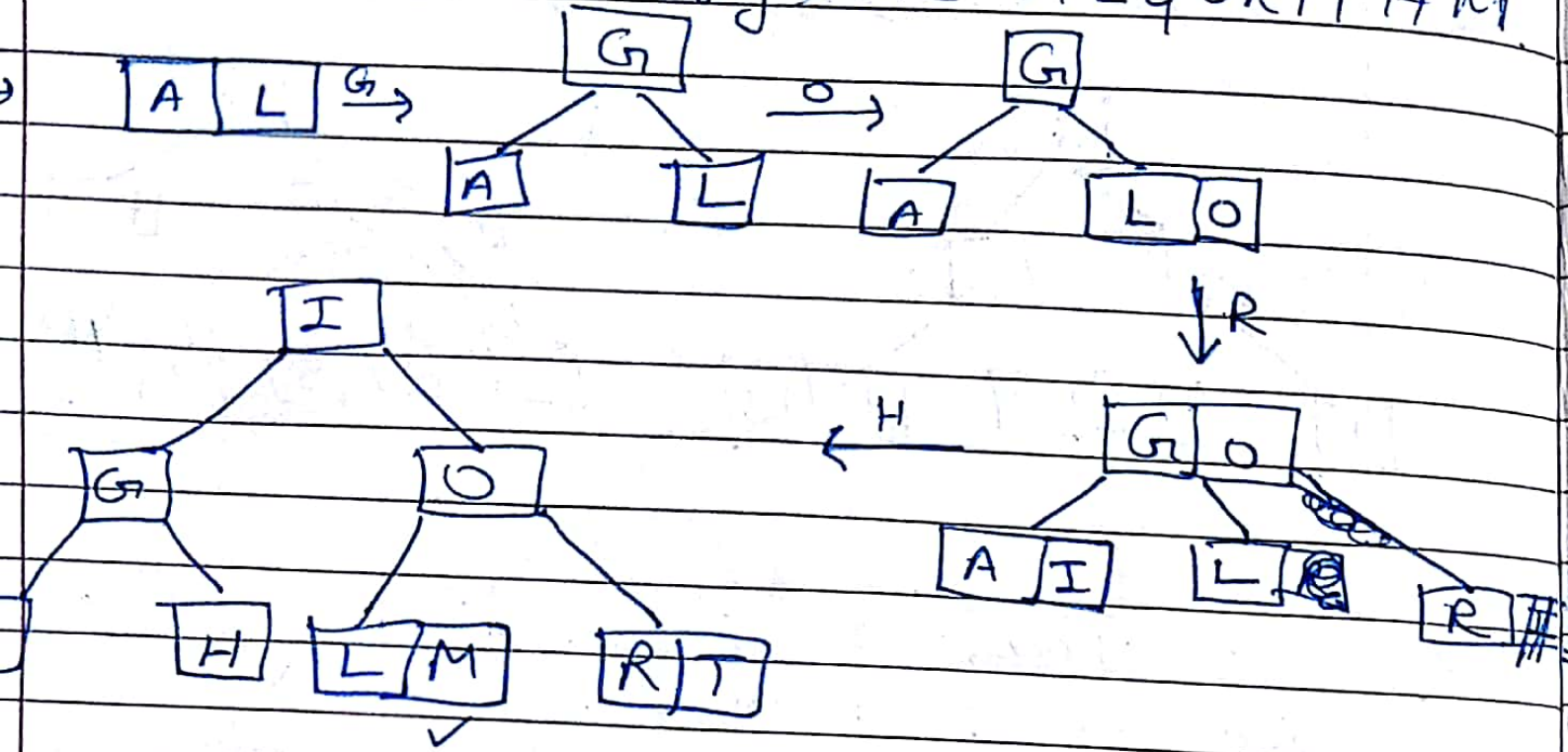
* height of tree 'h'

$\log_3 n - O(1) \leq \log_n n + O(1)$

* Every leaf node will have same level of tree root moves up ~~rather~~ rather than leaves moving down.

Page No.
Date

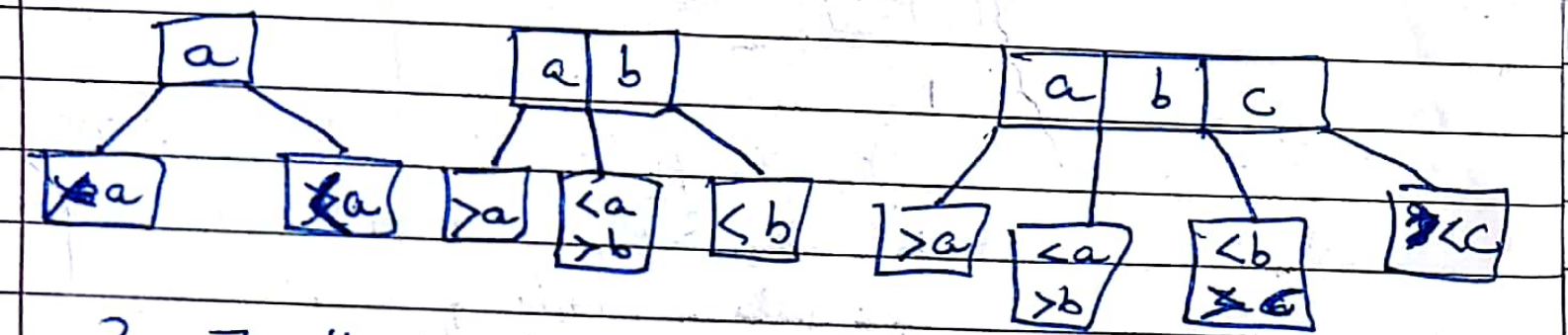
Create 2-3 tree using letter ALGORITHM.



2, 3, 4 Tree

4 children

Each node can have 1, 2 & 3 keys.
Every leaf will be at same level.

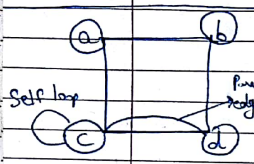


3, 7, 4, 9, 12, 0, 5, 6, 8, 2, 1

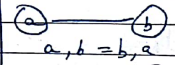
Page No.
Date

Graph Theory

Graph : A graph G is an ordered pair of a set V of vertices & a set E of edges.
$G = (V, E)$



* Undirected Graph

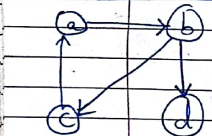


$a, b = b, a$
Unordered Pair

$G = (V, E)$

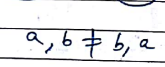
$V = \{a, b, c, d\}$

$E = \{(a,b)(b,d)(d,c)(c,a)\}$



* Directed Graph

Ordered Pair

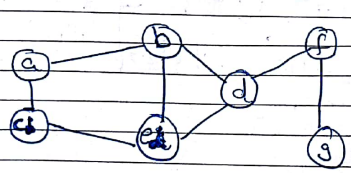


$a, b \neq b, a$

$G_1 = (V^d, E^d)$

$V^d = \{a, b, c, d\}$

$E^d = \{(a,b)(b,d)(b,c)(c,d)\}$



Walk : A sequence of vertices where each adjacent ~~pair~~ pair is connected by an edge.

Path: A path is a walk in which no vertices are repeated.

Cycle: $V_S = V_D$

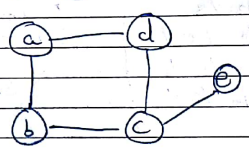
Degree of Graph : Sum of all the degree of vertices.

Complete Graph : All the vertices are connected to each other.

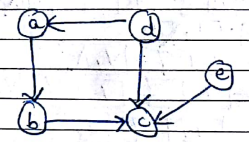
Trail : A walk in which no edges are repeated.

* **Representation of Graph**

– **Adjacency Matrix**



i	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	0
C	0	1	0	1	1
D	1	0	1	0	0
E	0	0	1	0	0



i	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	0
C	0	0	0	0	0
D	1	0	1	0	0
E	0	0	1	0	0

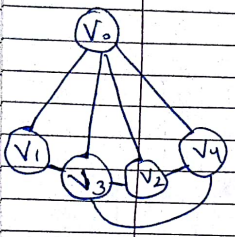
→ Path Matrix →

i	A	B	C	D	E
A	0	1	1	1	1
B	1	0	1	1	1
C	1	1	0	1	1
D	1	1	1	0	1
E	1	1	1	1	0

Page No.
Date

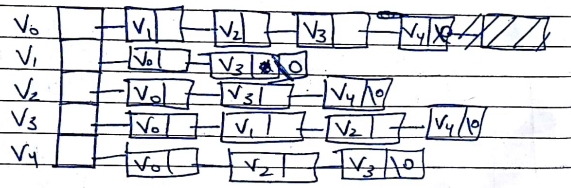
Adjacency List → More memory efficient than adjacency matrix

It consists of a list of vertices & are represented by either linked list or ~~array~~ array. For each vertex, adjacent vertices are represented in the form linked list.



Node	Adjacency List
V_0	V_1 V_2 V_3 V_4
V_1	V_0 V_3
V_2	V_0 V_3 V_4
V_3	V_0 V_1 V_2 V_4
V_4	V_0 V_2 V_3

```
struct node
{ int vertex;
  struct node *next;
} node
node *head [ ]
```
↓ Size of total nodes

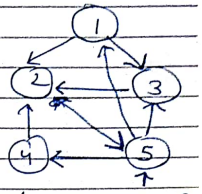


v → vertices
e → edges
n → nodes

$n = v + 2e$

* **Graph Traversal**

→ Visiting all nodes of the graph.
→ Two types:
– Breadth First Traversal (BFS)
– Depth First Traversal (DFS)



* **Breadth First Traversal (BFS)**

1. Create a queue Q.
2. Mark V as ~~&~~ visited. Put V in the queue.
3. While Q is not empty, ~~traverse~~ remove the ~~front~~ head u of Q & mark & enqueue all unvisited neighbours of u.

o/p of traversal: 5 1 3 4 2

Q: | 5 |

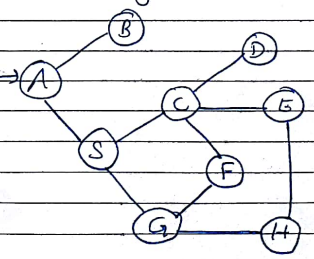
Q: | 1 | 3 | 4 | (F, R)

Q: | 3 | 4 | 2 |

Q: | 4 | 2 |

Q: | 2 |

Eg 2 →



Q: | A |

Q: | B | S |

A B S C G D E F H

Q: | S |

Q: | C | G |

Q: | G | D | E | F |

Q: | D | E | F | H |

Q: | E | F | H |

Q: | F | H |

Time complexity: E, V

$= O(E+V)$

$= O(|E| + |V|)$