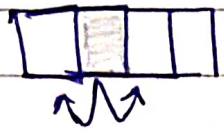
1st Now {Stack} - linear data structure
- one predecessor and one successor only



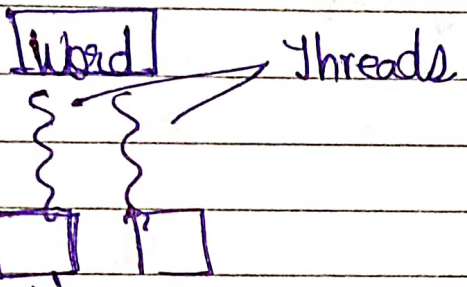
LIFO - Last In first Out

Applications - Recursions
Memory Management
Imp ✓ Expression Evaluation
Parantheses Matching

Backtracking { to check parantheses matching / testing in an expression }

App - can control memory allocation and deallocation
- helps to clean up the objects

① Memory Management -



Stack LIFO

Physical Memory

② Expression Evaluation

{Top = Size - 1}

→ (2+3)+x+y

↗ Overflow

3	50
2	20
1	10
0	8

Top {Current Position}

1. Push (Insert at last)
2. Pop → is_Empty top - = 1
3. Traverse

True "Under flow"

False

Page No:
Date: / /

Push → top = top + 1
insert
"top" should start from -1

```
int top = -1
int stk. array [4], x;
void push () {
    if (top == size-1) {
        printf ("Overflow")}
    else {
        top += 1
        stk_array [top] = x
    }
void main () {
    push();
}
```

size = 5

Page No:
Date: / /

2nd Now Pop

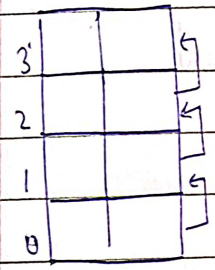
```
int isEmpty (struct Stack *ptr)
{
```

⇒ Check whether the array contain any element or not

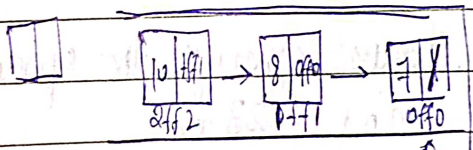
```
if (top == -1) {
    printf ("Underflow") }
else {
    printf ("%d" stk_array [top])
    top = top - 1;
}
```

Stack Implementation with Linked List

↳ no use of top in case of stack implementation on the singly linked list



Note



⇒ Always push and pop from the beginning.

First element of stack will the last of the linked list

Note – Stack need to implemented in array and Linked List

9th Nov "Types of parsing"

Expression Evaluation
→ Infix 2+3
→ Prefix + 23 (operator पहले)
→ Postfix 23+ (operator बाद में)

- Conversions of these three possible

- a+(b*c)-e
  operands operators

Infix - two I operator in between 2 operand
- eg a+b (2+3)

Prefix ⇒ operator comes first and then operand
eg +ab (+23)

Postfix - operator comes after the operand
eg ab+ (23+)

Application - When we do compiler design

-precedence of *,/ remain same

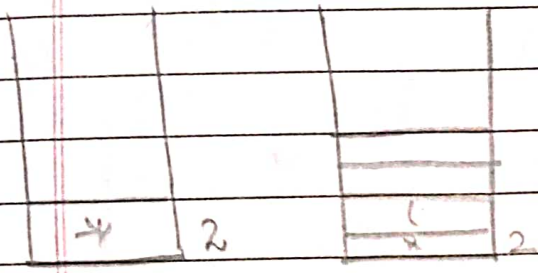
Evaluate

Q Solve the given Infix operations using Stack

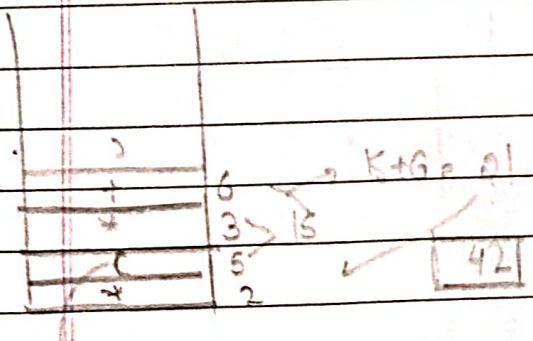
Page No:
Date: / /

1) Stack का उपर operator using precedence (LIFO)

⇒ 2*(5*(3+6))/15-2



∴ 2*(5*3+6)



no need to check precedence since bracket is open)

- Precedence
  1) ^
  2) * /
  3) + - %

- Highest priority उपर
  उपर है तो lowest priority can not go

- If opening bracket their then no need to check highest / lowest priority outside the bracket

- Inside the bracket need to check the precedence

- If same precedence in bracket then use LIFO

- At end, we get an empty stack of operator and an element in operand stack (which will be the answer)

*/
+ -

Page No:
Date: / /

2*(5*(3+6))/15-2

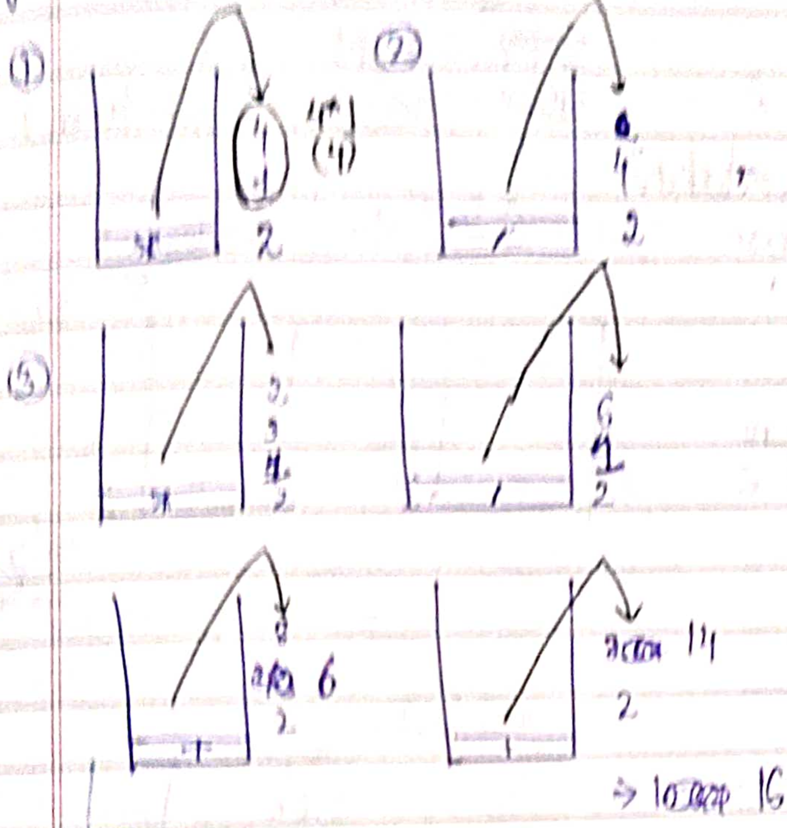
										+	
						(				(	6
						*	3			*	3
			(	5		(	5			(	5
*	2		*	2		*	2			*	2

)											
+											
(		6		)							
*		3		*		9					45
(		5		(		5		*			2
*		2		*		2					

Add can be their own
there is high priority (*)
because we have opening
bracket

Prefix → (right to left pure (move))

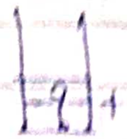
eg + + 8 / * 2 3 1 / * 4 1 2



① → scan right to left

+ab ⇒ a+b

②



पहले वाला right side, उसके बाद left side
a+b

③ No checking precedence

Parse	Push	Pop	Value
2	2		
1	1		
4	4		
*	*	*	4
/	/	/	2
1	1		
3	3		
2	2		
*	*	*	6
/	/	/	6
8	8		
+	+	+	14
+	+	+	16

* Parsing from right to left

→ ALU (Arithmetic Logic Unit) can solve only postfix

→ Result remain same in infix, prefix and postfix

Page No:
Date: / /

Prefix +23 ⇒ 5

2+3

+9 * 26 → Main expression

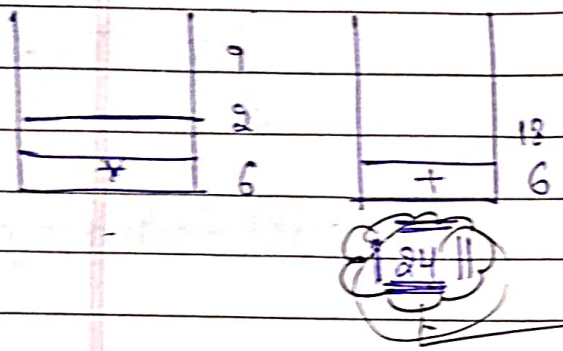
Method 2

Prefix - right to left (Precedence not taken)

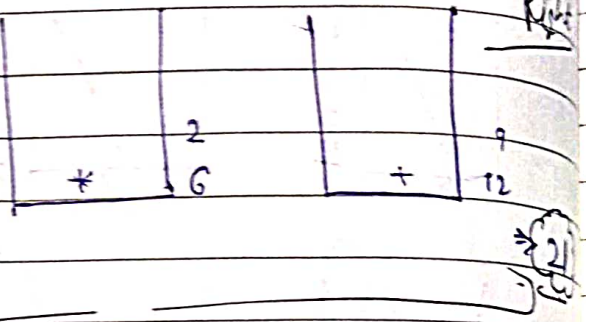
① Parse right to left (Precedence taken)

WRONG

62*9+ — reverse

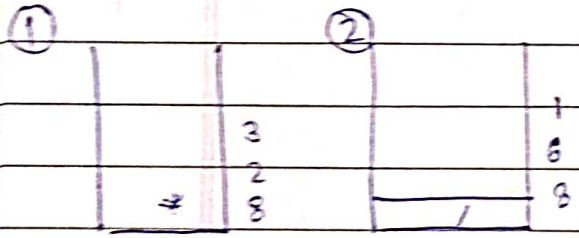


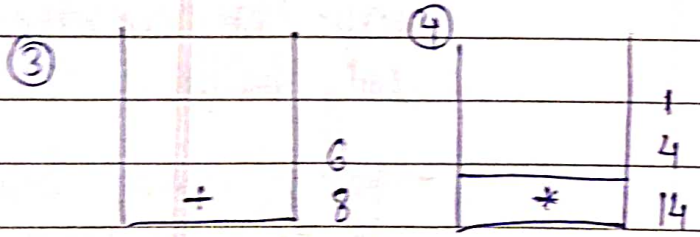
62*9+ — reverse

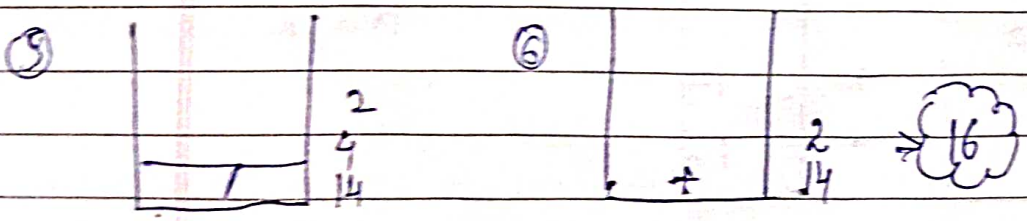


Postfix ⇒ left to right parse

823+1/+41*2/+



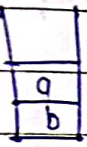




* Prefix ←

+ab

a+b



* Postfix →

ab+

a+b

Precedence → Associativity left to right

Imp

Page No:
Date: / /

Infix to Prefix

A+B * (C+D) / f + D * E

↙ operand

A+B * +CD / f + D * E

A+ * B+CD / f + D * E

A + / * B+CD f + D * E

A + / * B+CDf + * DE

+A/B * B+CDf + * DE

+ +A/B * B+CDf * DE

→ Note

* Prefix ⇒ Right to left
* Postfix / Infix ⇒ left to Right
* Infix - Precedence in account
  Postfix / Prefix - not in account

Page No:
Date: / /

Queue (FIFO)

Insertion

Overflow - Rear = Size - 1

Underflow → Rear = front = -1 (when created the queue)
Front > Rear

Deletion

{ Enqueue → Rear
Dequeue → front }

Insertion ① Check overflow
② Increment rear then value insertion

→ If (rear == (Max - 1)) OR (Size - 1)
printf ("Overflow");

→ If rear = front = -1; { when the queue is inserted first time }
Front = 0;
Rear = 0;

→ else;
Rear = Rear + 1
Queue [Rear] = data;

Dequeue → If (front = Rear == -1) // (front > Rear)
printf ("Underflow")
else
val = queue [Front]
front = front + 1

Application
→ Data Packets in Data Network flow (Handling website traffic)
→ Maintaining playlist in Media-Players
→ CPU scheduling and disk

Linked List

1. Insertion from end ⇒ Enqueue
2. Deletion from front ⇒ Dequeue

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

void enqueue (int *data, struct node *head){

    struct node* new_node = (struct node*) malloc
                            (sizeof(struct node*));
    new_node→ data = data
    new_node → next = NULL;
    while (head → next != NULL){
                head = head→ next ;}
    head
```

इसमें rear से front पर भी जाते हैं।
In circular no rear / front fast
It follows FIFO

Page No:
Date: / /

Circular Queue

- int isfull() {

```
if ((front == rear+1) || (front == 0 && rear == size-1))
    return 1
else
    return 0
}
```

```
int isempty() {
if (front == -1)
    return 1
else
    return 0
}
```

- void enqueue (int item) {

```
if (isfull())
    printf("Overflow")
else
    if (front == -1)
        front = 0
    else
        rear = (rear+1) % size
    cir-Queue [rear] = item
}
```

eg 

1%5

```
int dequeue () {
if (isempty())
    printf("Underflow")
else
    item = Cir_Queue [Front]
if (front == rear)
    Front = -1
    rear = -1
else
    Front = (Front+1) % size
return item;
}
```

Insertion from Rear + Front
Deletion from Front + Rear

Page No:
Date: / /

Deque / DE-Queue

→ Doubly-Ended.

→ do not follow FIFO

→ called queue because can add/insert both from front/end but not from the middle

Input restricted queue

- Enqueue - Rear
- Dequeue Rear/Front

Insertion from front not allowed

Output restricted queue

Enqueue - Rear/Front

Dequeue - Front

Deletion from rear not allowed

Enqueue - Front, Rear
Dequeue - Front, Rear

front = -1
rear = 0

Enqueue

```
Enqueue_front ()
    if (front < 1)
        front = n-1
    else
        front = front - 1
    dequeue[Front] = item;
```

IF
DR

Right to left

enqueue - rear and dequeue - front

H.W

Application

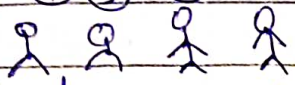
Right to left

```
Dequeue_rear ()
① front = -1
      Underflow
2. front == rear
      front = -1
      rear = -1
3. rear == 0
      rear = n-1
④ rear = rear - 1
```

Page No:
Date: / /

• Queue ⇒ FIFO

⇒ ④ ③ ② ①



Insertion end — Deletion end

Queue ADT

⇒ Storage — Methods - Enqueue - Peak
Ins. End - Dequeue - isEmpty
Del. End - first val - isFull
- last val

Queue → Application

Insert → Increment back end] O(1)
(enqueue) → Insert at back

dequeue → ① Remove element at 0] O(n) This we find other alternative
② Shift all elements maintain a front index and it will not contain index of first element but index of prior to first element

Initially Front and back = -1

Queue empty ⇒ front = back
Queue Full ⇒ Back = size - 1

Merge Sort → O(n log n)

Page No:
Date: / /

- work on 'divide and conquer' strategy
- Recursive Fund^n used
  - ↳ call itself
  - ↳ starting fund^n - base fund^n
  - ↳

eg
```
sum (int n)
{ if (n==0)
      return n; }
  else
  { return n+ sum(n-1)
  }
```

Divide and Conquer

① 5 + sum(5) ⑪ ↩
② 5 + sum(5-1) ⑪ ↩
③ 5 + 4 + sum(4-1) ⑪ ↩
④ 5 + 4 + 3 + sum(3-1) ⑪ ↩
⑤ 5 + 4 + 3 + 2 + sum(2-1) ⑪ ↩
⑥ 5 + 4 + 3 + 2 + 1 + sum(1-1) ⑪
(1-1 → base condition)

* ⑪ Pause

Note · divide till atomic problem

eg
1st [9|7|3|6|2] (0 1 2 3 4)
2nd [9|7|3] [6|2]
3rd [9|7] [3]
4th [9] [7]

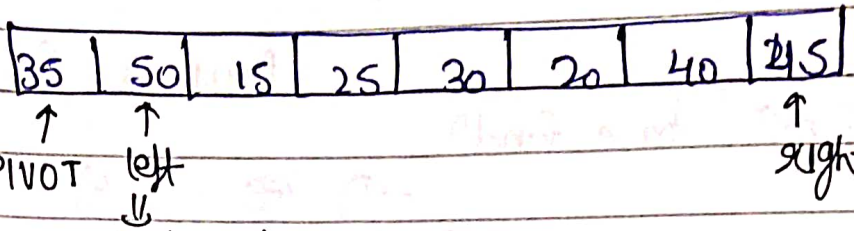
* m = (l+r)/2
* divide = recursive
merge ⇒ next algo

→ learn Flowchart

Page No:
Date: / /

1. After first pass, pivot element reaches at right position

2



Final pivot
= 45 element

3

Page No:
Date: / /

Quick Sort

Pseudo Code

```
int pivot_position (arr, first, last)
{  pivot = arr[first]
   left = last + 1
   right = last
   while (True) {
```

- left and right not crossing each other → `while (left <= right && arr[left] <= pivot)  // move left`
  `left = left + 1`
  `while (left <= right && arr[right] >= pivot)`
  `right = right - 1  // move right`
- right and left cross each other `if (right < left)`
  `break  // यदि if को break`
  `else`
- left and right swap `swap (arr[left], arr[right]) }`
  `swap (arr[first], arr[right])`
  `return right`

```
}
void quicksort (arr, first, last) {
    p = pivot_position (arr, first, last)
    quicksort = (arr, first, p-1)
    quicksort (arr, p+1, last)
}
```

Difference → internal / external
→ pivot / mid way
→ time complexity
→

Time Complexity

Sorting Algo	Best (Ω)	Average (Θ)	Worst (O)	Space Complexity
Bubble	n	n^2	n^2	O(1)
Selection	n^2	n^2	n^2	O(1)
Insertion	n	n^2	n^2	O(1)
Merge	nlogn	nlogn	nlogn	O(n)
Quick Sort	nlogn	nlogn	n^2	O(n)

⟹ Here need to Specify that best case in Quick Sort : when pivot element in middle

Note
- When data not large, then need to use → Bubble, Selection, Insertion
- Work on Divide and Conquer → Merge, Quick

- Sorted array best case for all the sort except Quick sort

In Quick sort
- Worst case when array sorted $O(n^2)$
- Best case when pivot element come in middle $O(n \log n)$

- When data large then need to use → Merge, Quick

Request → Hit / Miss

Cache - much faster than RAM
- directly accessed by CPU

Page No:
Date: / /

- When data not huge and sorted array passed then which sorting algo to use?

< Bubble ✗ Ω(n)
Insertion ✓ Ω(n)
(why?)

→
Insertion Sort → Cache Miss ↓ → swapping at end of i loop
Bubble Sort → Cache Miss ↑ → swapping takes more frequently

Q When data uncertain can not find exact element and data very huge → Merge Sort

Whole
Note - Data not transffered to RAM / Cache when program called. Small chunk of it only transffered.
When data required then CPU checks in cache if it find that data it is called cache hit and if data not transffered then it is called cache miss.
If cache miss then CPU alert and tell cache to upload more data.

{ concept of Page segmentation }

→ Balancing

Page No:
Date: / /

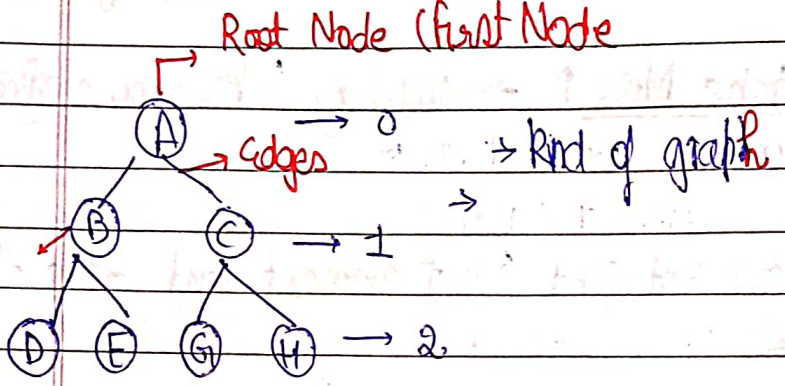
24th Dec

Trees → Non Linear Data Structure (more than one predecessor or more than one successor) → Graph

→ special kind of graph

- General Tree
- Binary Tree
- Aborant Tree

* generally through linked list



Successor = 2
Predecessor = 1
} Hence Non linear Data Structure

Terminology

1) Node
2) Edge
3) Root (First element)
4) Parent Node
5) Child Node
6) Sibling { One parent of two eg B, C }
7) Leaf Node { having no child } eg D, E, G, H

8) Degree
- In – अंदर आने edge
- Out – बाहर जाने edge

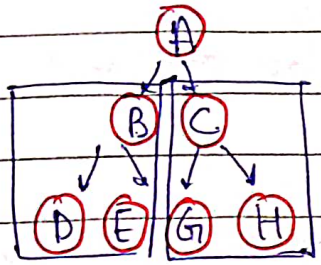
eg B → In degree = 1
Out degree = 2

Page No:
Date: / /

- No. of edges from root node to given node = Depth of that Node

a) Level - Root Node start with level 0

b) Subtree → When a tree can be broken down into subsequent trees



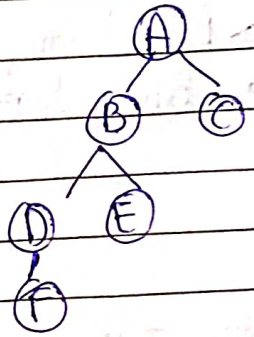
Application ① Compiler Design - Parsing, Parenthesis Management can be done in more optimized way as compared to stack

② Networking - shortest path using tree and graph

③ Cloud Computing → Cloud datacentre's data stores in cloud computing Fashion

④ Deep learning / Machine learning →

Properties of Tree → a) No. of nodes = N
No. of edges = N-1



b) Tree kind of graph with no loops / circuit

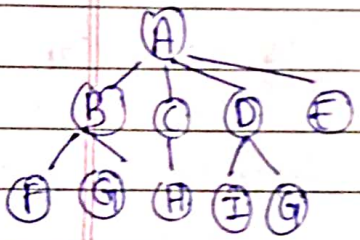
c) Depth - Particular to node (no of edges from root node to given node
⇒ eg D → 2 → Depth

d) Height - Longest possible path of the graph tree
Height = 4 (A < B < D < F)

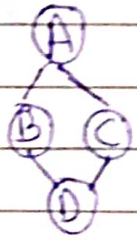
5) Degree of Tree → max no of child of that particular parent
→ max. no of child node in tree
eg In given tree = 2

→ generally in Tree In degree = 2 - less significant

Types of Tree
① General Tree - (no condition on no of child, left node, root node)
→ child can be any no. of times but parent need to be one



General Tree



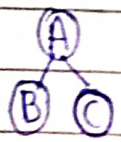
Not General Tree - loop - more than 1 parent

② Binary Tree
- Full Binary Tree (Strict Binary Tree)
- Complete (Proper) Binary Tree
- Almost Complete Binary Tree (ACBT) - related to Heap
- Binary Search Tree (BST)

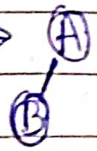
- Almost two children (0,1,2) ✓
- Parent need to be one

Binary Tree eg →

→

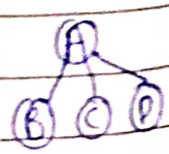


→



→

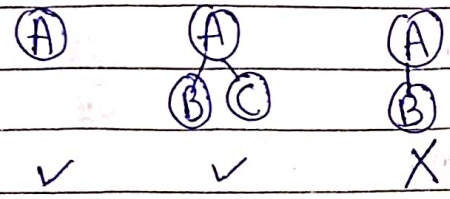
Not Binary Tree



Page No:
Date: / /

Full Binary Tree → या तो 0 children OR 2 children (should not be 1 child)

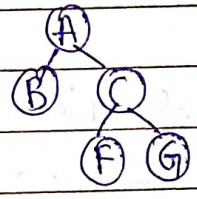
eg



Complete Binary Tree – एक level से दूसरे level जाने के लिए पहला fill होना
– एक level पूरा fill होना चाहिए
– left to right fill and when one level completely filled then move to next

eg Fully और sare level fill
Complete and full Binary Tree

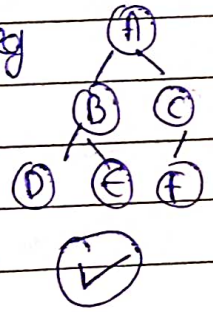
eg

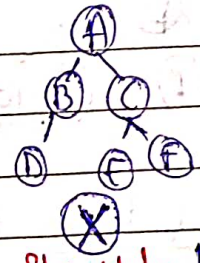


Full Binary Tree but not complete binary Tree

Almost complete binary Tree
- mostly implemented in Heap
- level पूरा होना जरूरी नहीं
- left to right entry

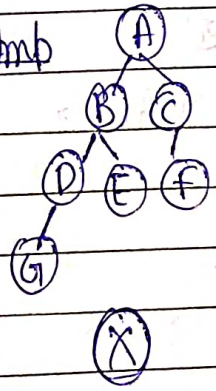
eg





Should be left to right

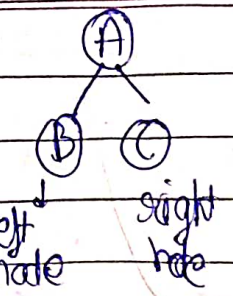
Amb



Not Full

Page No:
Date: / /

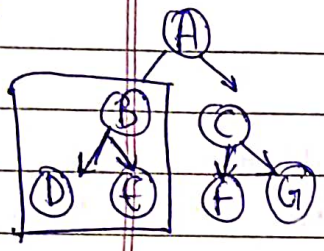
Traversal of Binary Tree



1. Preorder — Root, left, right
2. Inorder — left, Root, right
3. Postorder — left, right, root

Note: Depend on "positioning of ~~Binary~~ # Root"

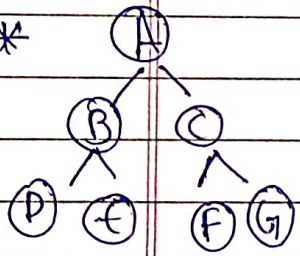
*



* {Preorder Traversal} → A B D E C F G

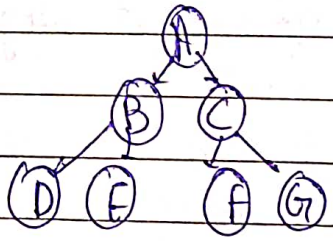
full left traversal

*



In ~~Post~~ Order Traversal = D B E A F C G

* Post Order



D E B F G C A → Root

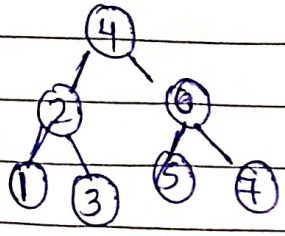
left right

Binary Search Tree

- left < Root < Right
  Value Value Value
- Type of binary tree

eg form BT

4 2 3 6 5 7 1 (Need more data)



Page No:
Date:

Inorder Traversal • left root right

1 2 3 4 5 6 7

{BST Traversed ascending}

Note → Binary Tree → O(n) → No restriction
→ Binary Search Tree → O(log n) → left < root < right

29th Dec 2022

Traversal → Tree

minimum two combination given

• In → DBEAFC (Use to find left and right element)
Pre → ABDECF {Use for root element}

Q1 First to draw the graph then Find post order DEBFCA

Root

① In - DBEAFC (LRoR)
Pre - ABDECF (RoLR)
Root

① A
DBE FC left
In - F, C
Pre CF
Root

② A
left DBE C
In - DBE F
Pre - BDE
Root

③ A
B C
D E F

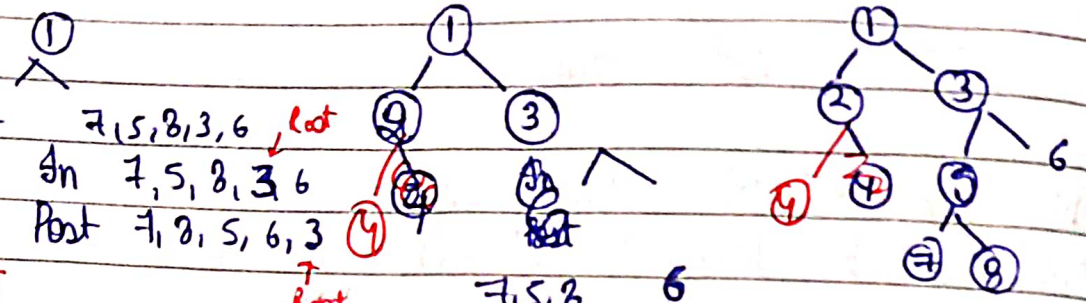
Ro LR

In - ~~Root~~ Left |
Post - ~~Right / left~~ Root

Page No:
Date: / /

Q2 In = 4, 2, 7, 5, 8, 3, 6 (L Ro R)
Post = 4, 2, 7, 8, 5, 6, 3, 1 (L R Ro)

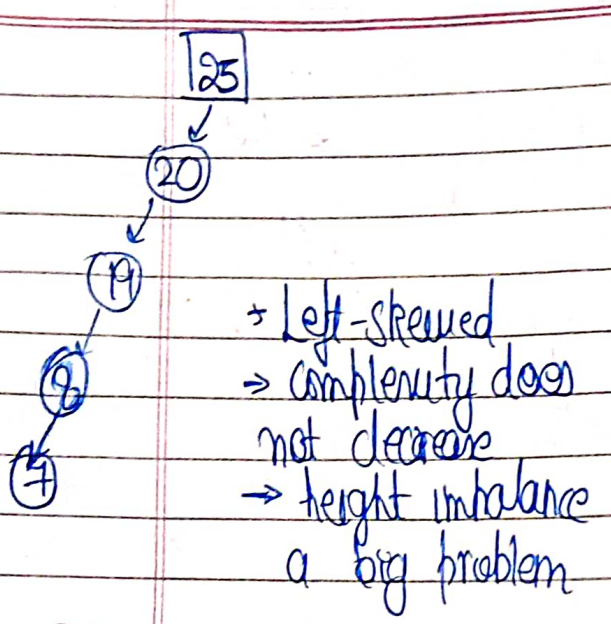
Left Right Root



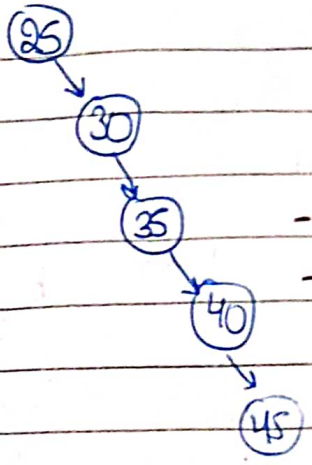
Pre → {1 2 4 3 5 7 8 6}

Post
Pre

5th Jan 23



- Left-skewed
→ complexity does not decrease
→ height imbalance a big problem

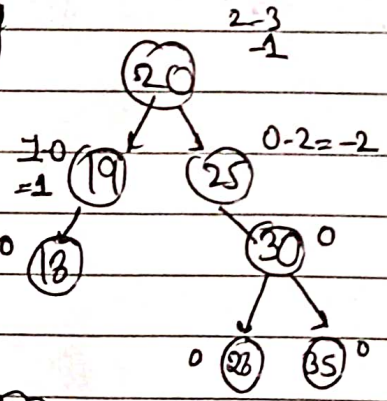


- right skewed
- does not get benefit of Binary search tree

AVL → Height balanced tree

① Balance factor ⇒ Height of (LST) - Height of (RST)

of every node eg



② BF (Balance factor) → 0, 1, -1

If BF of all node in {0, 1, -1} then tree balanced

Creation of AVL

Rotation Rules

1. Left to right parse
2. Check BF each step

Imbalance

Rotation
RR → L left
LL → R
LR → RL
Interchange

Note: After rotation, tree should be balanced and follow BST

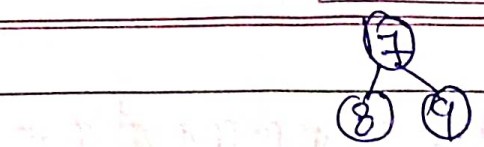
If any node left out in between rotation check with help of BST

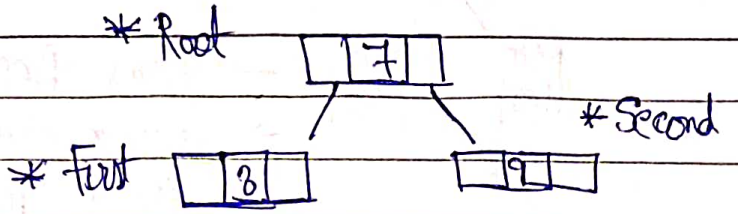
Name → always in linked list

Page No:
Date: / /

"Tree Implementation"

```
struct node {
    int data;
    struct node * left
    struct node * right;
}
```





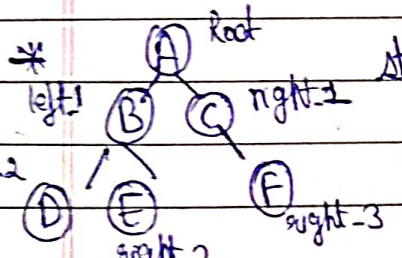

```
struct node * Root
struct node * left;
struct node * right;
```

{Memory Allocate}

```
Root → left = first
Root → right = second

first → left = NULL
first → right = NULL

second → left = NULL;
second → right = NULL;
```




```
struct node {
    char data
    struct node * left
    struct node * right;
}

struct node * Root
struct node * left_1
struct node * right_1
struct node * left_2
struct node * right_2
struct node * right_3

Root → data = 'A'
Root → left = left_1
Root → right = right_1

Root left_1 → data = 'B'
left_1 → left = left_2
left_1 → right = right_2

right_1 → data = 'C'
right_1 → left = NULL
right_1 → right = right_3

left_2 → data = 'D'
" → left = NULL
" → right = NULL
```

Date: / /

10th Jan "Heap Sort"

1. ~~BST~~ Heap → implementation in graph (spanning tree

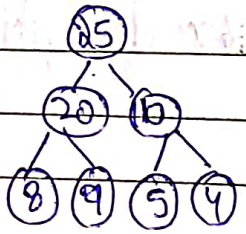
Heap
- Min Heap
- Max Heap

→ Uses ACBT
→ Common operation
(i) Insertion (iv) Deletion
(ii) Sort

Min Heap → follows ACBT (left to right, one level complete) → not a type of BST
→ max 2 child
→ Value of parent < Value of child (Minimum)
[(left can be greater than right) no relation in sibling]

Max Heap → Value of parent ≥ Value of child (Max)

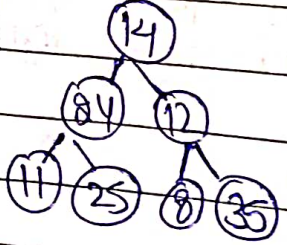
→ eg



Insertion ① One by one
② Heapify (Building Heap)

eg 14, 24, 12, 11, 25, 8, 35

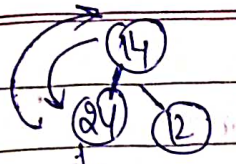
① Simple ACBT



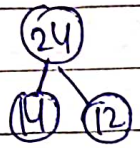
'One by One Method'

Page No:
Date: / /

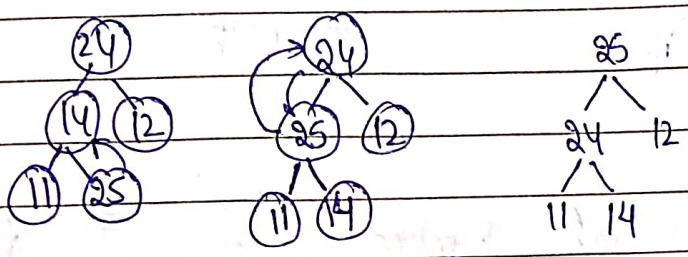
Step1 Check whether follow max heap



replace max of the child node

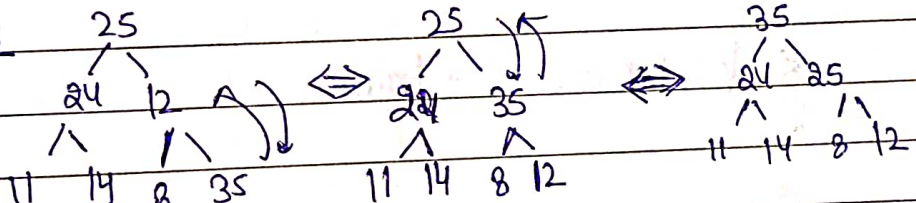


Step2



Not max heap

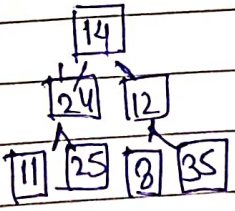
Step3



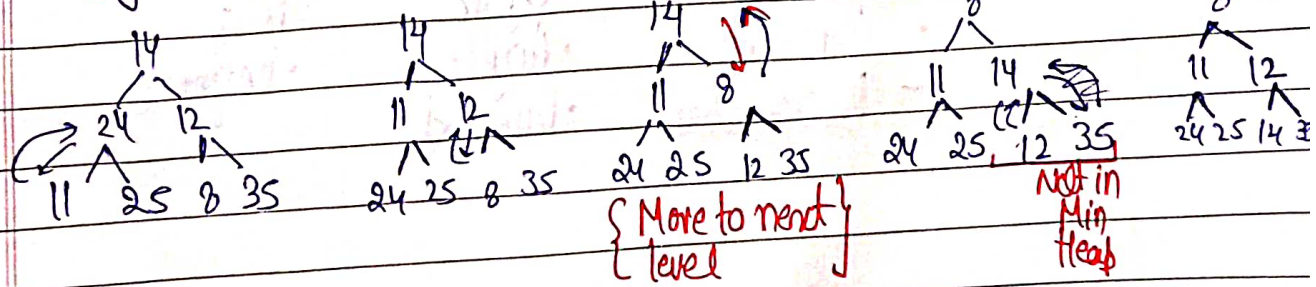
* Need to swap only data not node, because if swap node then left and right will change

Min Heap ("Heapify")

Step1 Create {BFS}



Step2 { Bottom to top, left to right }



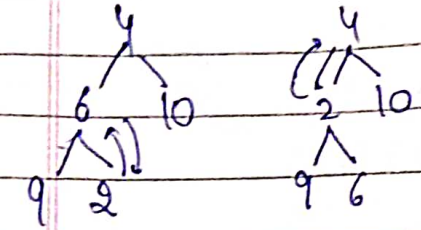
{ More to next level }

Not in Min Heap

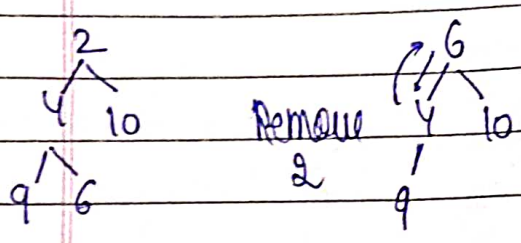
Page No:
Date: / /

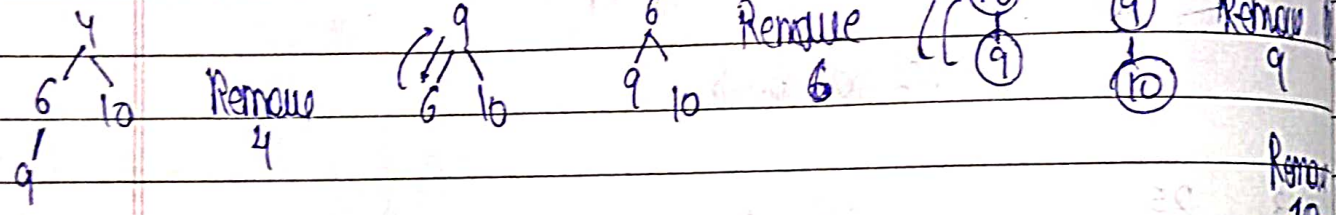
Heap

Sort · 4, 6, 10, 9, 2

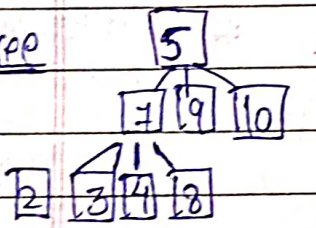


Step1 Create min heap
Step2 Then delete the root element after printing
→ we will replace last level right most node to node
(so that ACBT remains) and then again make in min heap





M-tree



* children are uncertain
*

Graph → non linear data structure
→ used to find shortest path
→ very much use in blockchain
→ used in computer design

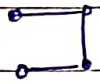
G(V, E) · finite · Trivial · Simple
· Infinite · Cyclic · Multi
· Connected · Acyclic · Weighted
· Disconnected · Directed

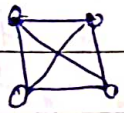
(Hashing Preferable)

* Tree type of graph without cycle

Infinite 

{Complete Graph} - P2P connection - every node has 'n-1' degree

Connected 



Trivial • (without edge only 1 vertex)

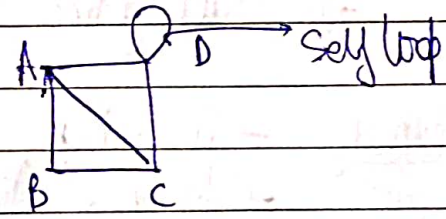
Cyclic • If even one cycle present then cyclic

Simple – Only 1 edge btw two vertex

Weighted → if each vertex is assigned with a weight (either cost)

Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	1

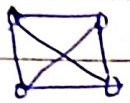
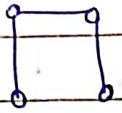
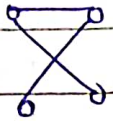


If directed then

→ If weighted graph then in place of 0 and 1 we will write weighted value.

Page No:
Date: / /

Spanning Tree → shortest distance path
① Need to take out a subgraph such that
(a) Include all the vertices
(b) No of edges = V − 1
(c) connected
(d) no cycles (tree cannot be cycle)

eg  Spanning Tree  

Note- Without a graph, we can generate many no of spanning tree
$= n^{n-2}$ {n = no. of vertices}

∴ Spanning tree with least cost will be best tree

Min Spanning Tree → Prim's Algo } work on
→ Krushal Algo } Greedy Approach

Prim's Algo - start with any one vertex
- mn cost remain same (structure of min. spanning can be different)

Min Spanning Use → broadcast a ~~not~~ packet in a network
→ water supply

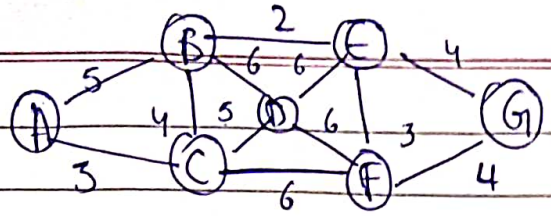
Intermediate graph can be disconnected

Min spanning tree → min cost guaranteed

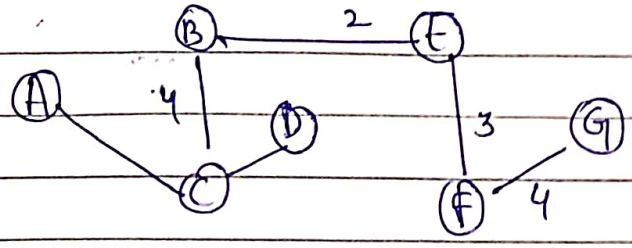
Page No:
Date: / /

Sort all the edges

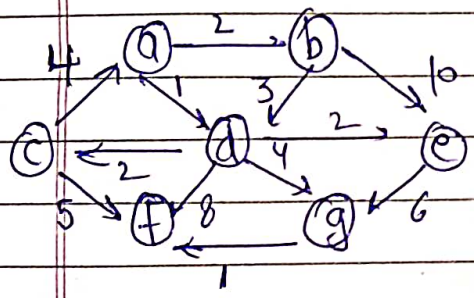
Krushal Algo



① Draw all vertex
② Then choose min weighted edge



Dijkstra → - Unexplored →
- explored → which we have explored



* Let 'a' be my source

	Explored	Unexplored
1.		$a^0, b^\infty, c^\infty, d^\infty, e^\infty, f^\infty, g^\infty, t^\infty$
2	a^0	$b^{2a}, c^\infty, d^{1a}, e^\infty, f^\infty, g^\infty, t^\infty$
3	a^0 d^{1a}	b^{2a} c^{2d} e^{2d} f^{8d} g^{4d} b^{2a} c^{3d} e^{3d} f^{9d} g^{5d}
4	a^0 d^{1a} b^{2a}	c^{3d} e^{3d} f^{9d} g^{5d}
	a^0 d^{1a} b^{2a} c^{3d}	e^{3d} f^{8c} g^{5d}
	a^0 d^{1a} b^{2a} c^{3d} e^{5d}	f^{8c} g^{5d}
	a^0 d^{1a} b^{2a} c^{3d} e^{5d} g^{5d}	f^{6g}

* $1a < 2a$

* can choose any c/e

a	
b	a → b = 2
c	a → d → c = 3
d	a → d = 1
e	a → d → e = 3
f	a → d → g → f = 6
g	a → d → g = 5