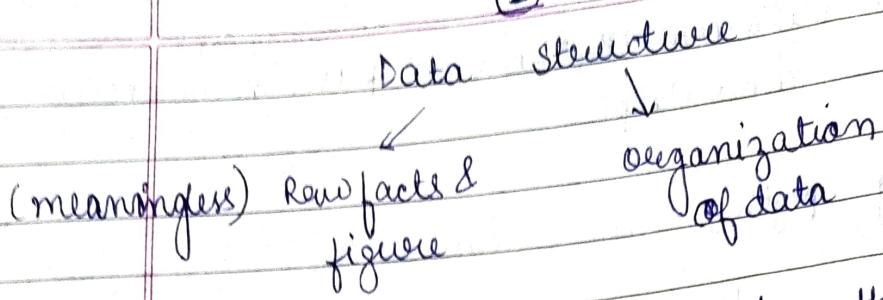


Complete DSA Notes

— By Riti Kumari

(1)



Data structure is logical and mathematical model of storing and organizing data in a particular way that is can be required for designing and implementation of algorithm.

e.g: Array, linkedlist, stack, queue.

Problem

~~datastructure~~ Algorithm

Program → C/C++

Data → Raw facts

Information → Meaningful data

(2)

Types of DS

Data structure

~~Primitve DS~~

Primitive DS

- 1) int
- 2) char
- 3) float
- 4) double
- 5) boolean

~~Non primitive DS~~

Non primitive DS

linear

Array

linked list

Stack (LIFO)

Queue (FIFO)

Non linear

Tree

Graph

Primitive DS - DS that directly operate upon the machine.
These are predefined operation & properties

Non primitive DS - Derived from primitive and not directly work upon machine.

Linear DS •

- 1) All elements are arranged in linear order where each element had successor and predecessor except first & last element.

2) Single level involved.

3) Used in S/W development

Non linear DS

This DS doesn't form a sequence. Data element are arranged hierarchical.

2) Multilevel involved.

3) Used in AI.

(3)

Operations on DS.

DS operations

- 1) Traversing - accessing each record exactly once
- 2) Searching - finding location of the record with given key.
- 3) Inserting - Adding new record of DS
- 4) Deleting - Removing a record from DS.
- 5) Sorting - Arranging the record in some order

6. Merging - Combining 2 diff. sorted file into single file.

ADT (Abstract data type)

ADT refers to a set of value associated with operations or functions.

With ADT we know what a specific data type can do but how it is actually doing is hidden.

(4)

Time-space trade off

Time-space is way of solving problem

- 1) In less time by using more memory
- 2) In less memory but using more time.

(5)

Design and Analysis of Algorithm

↓
checking
performance

(Time &
space)

Problem



Algorithm + DS

↓ ↗
Program



i/p → Computer → o/p

Algorithm - It is a finite set of instruction if followed accomplish a task.

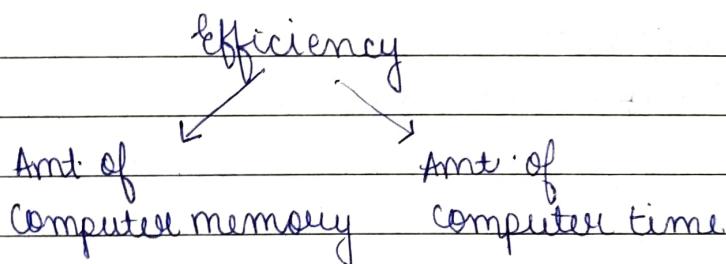
Criteria for Algorithm

- 1) Input
- 2) Output
- 3) Definiteness \rightarrow clear & unambiguous steps (b/o, a/o)
- 4) Finiteness \rightarrow Algorithm should terminate after few steps
- 5) Effectiveness \rightarrow (Time & space)

(5)

Analyzing Algorithms

Analyzing algorithms is require to detect the correctness and measure the efficiency of Algo.



There are 3 types of analysis

- 1) Worst case : Maximum no of steps taken on an any instance of size n.
- 2) Best case : Minimum
- 3) Avg case : Avg

(1) Sub(a, n)

$\{ \quad \quad \quad 0$
 $s = 0 \quad \quad \quad 1$
 for $i = 1 \rightarrow n$ do $\rightarrow n+1$
 $s = s + a[i] \rightarrow n$
 between $s - 1$
 $\} \quad \quad \quad x+n$

 $O(n)$

(2) Product(a[1...n], b[1...n, 1...p])

for $i \leftarrow 1 \rightarrow m$ do. $\rightarrow m+1$
 for $i \leftarrow 1 \rightarrow p$ do. $\rightarrow m(p+1)$
 $c[i, j] = 0. \rightarrow 1$
 for $k \leftarrow 1 \rightarrow n$ do. $\rightarrow mp(n+1)$
 $c[i, j] \leftarrow c[i, j] + a[i, k] * b[k, j]$
 between $c \quad \quad \quad \hookrightarrow mpn.$

 $O(mp n)$

(9)

Space complexity Analysis

Amount of memory needs to run to completion.

Space complexity $S(P) \rightarrow$ Constant space + Auxiliary Space

\diagdown
 i/p, local variable
 \diagup
 temp. variable

abc(a, b, c)

{

return a + b + b * c + (b + b - c) / a + b + 4 * 0

}

$$S(p) = 1 + 1 + 1 = 3$$

$$\rightarrow S(p) = O(1)$$

2. sum(a, n)

{

$$S = 0$$

for i = 1 to n do

$$S = S + a[i].$$

return S;

}.

$$S(p) = (n * 1 + 1 + 1) + 1$$

$\downarrow \quad \downarrow \quad \downarrow$

a S. i

$$= (n + 2) + 1$$

\times

we don't take in account
const. space.

$$= O(1)$$

3. def Rsum(a, n)

if n <= 0

return 0

else

return a[-1] + Rsum(a[-2], n-1)

$s(p) = \text{no of stack frames} * \text{space per stack frame}$

$$s(p) = n * \text{size of}(a) + \underbrace{\text{size}(f(n)) * \text{size of } a(n)}_{\text{const}}$$

$$= O(n)$$

(10)
Asymptotic notations
(Growth of a function)

Growth of funcⁿ:

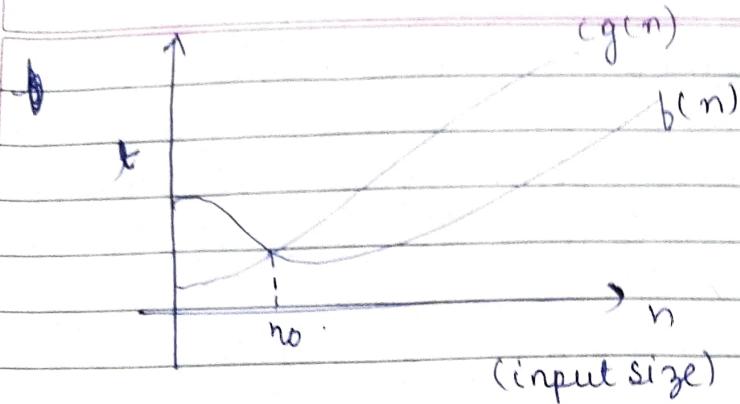
$$1 < \log n < \sqrt{n}, n < n \log n < n^2 < n^3 \dots 2^n < 3^n < \dots < n^n$$

Asymptotic Notations

- 1) Big-oh (O) \rightarrow upper bound
- 2) Big Omega (Ω) \rightarrow lower bound
- 3) ~~Theta~~ Theta (Θ) \rightarrow Avg Bound. (type bound)
- 4) Small-oh (o)
- 5) Small Omega (o)

Asymptotic notation are mathematical tool to represent complexity in terms of time and space.

- 1) Big Oh(O): The funcⁿ. $f(n) = O(g(n))$ if there exists positive constant c and no such that $f(n) < c \cdot g(n)$ $\forall n, n \geq n_0$.



$$f(n) = 3n + 2$$

$$g(n) = n$$

$$f(n) \leq c \cdot g(n)$$

$$3n + 2 \leq c \cdot n$$

$$c = 1 \quad \times$$

$$c = 2 \quad \times$$

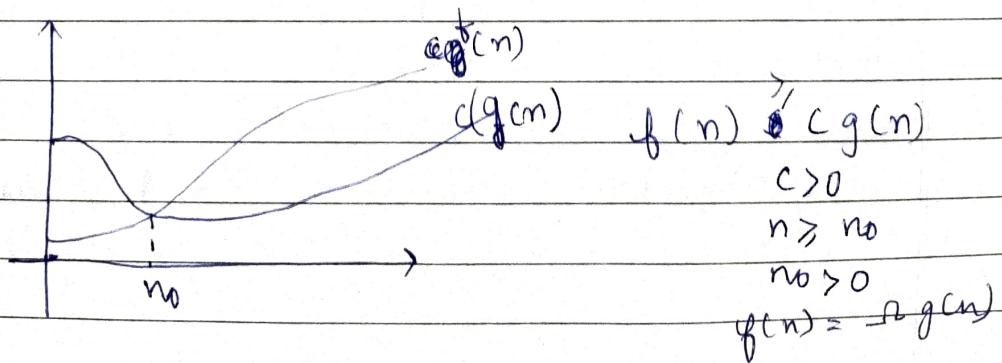
$$c = 3 \quad \times$$

$$3n + 2 \leq 4n \quad c = 4 \quad \checkmark \quad n_0 = ?$$

$$n_0 \left\{ \begin{array}{l} n=1 \times \\ n=2 \checkmark \\ n=3 \checkmark \end{array} \right.$$

$$c = 4 \quad n_0 = 2$$

2) Big Omega (Ω): The function $f(n) = \Omega g(n)$ if there exists a +ve constant c and n_0 , such that $f(n) \geq c g(n)$ for all n , $n \geq n_0 & > 0$



$$f(n) = 3n+2 \quad g(n) = n$$

$$f(n) > c \cdot g(n) \quad c > 0 \quad n \geq 0$$

$$3n+2 > 3n$$

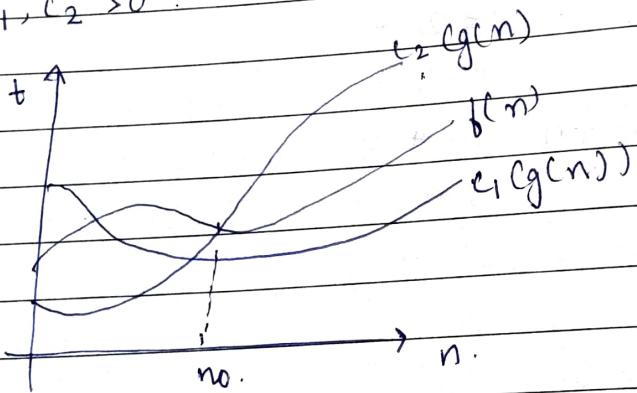
$$c = 3 \checkmark$$

$$n = 1 \checkmark$$

$$n = 2 \checkmark$$

$$f(n) = \Omega(n) \quad c = 3 \quad n \geq 1$$

3. Theta(Θ): The fun. $f(n) = \Theta(n)$ if there exists a +ve constant c_1, c_2 and no such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$
 $c_1, c_2 > 0$.



$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$c_1 n \leq 3n+2 \leq c_2 n$$

$$c_1 = 3$$

$$n \geq 1$$

$$c_2 = 4$$

$$n \geq 2$$

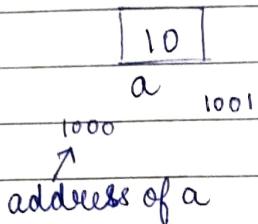
$$f(n) = \Theta(n) \quad \text{for } c_1 = 3, c_2 = 4, \quad n \geq 2$$

(11)

Array - It is a collection of similar type of data
eg - int, float etc.

This collection is finite and stored at adjacent memory location.

```
int a = 10;
```



Element - item / data stored

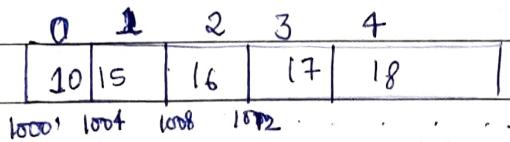
in array

index - location of element
from 0th - n-1th

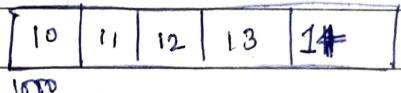
Address - numerical value of

1st byte at which
item is stored

```
int a[5];
```



base address.



$$a[0] = 10$$

$$a[3] = 13$$

$$a[1] = 11$$

$$a[4] = 14$$

$$a[2] = 12$$

$$\text{base address} = 1000$$

$$\text{address of } a[i] = \text{base address} + \text{index} * \text{size of element}$$

(12)

Array Types

- 1) 1D array
- 2) 2D array
- 3) 3D array
- 4) n-dimensional Array

One Dimensional array - An array which has one subscript is called 1D Array.

`int a[5] = {10, 20, 30, 40, 50};`

1000	1001	1002	1003	1004
10	20	30	40	50
index 0	1	2	3	4

How to find location of any array element

$$\text{loc } A[i] = b + w(i-l)$$

$i \rightarrow$ element whose address to be found

$b =$ base address

$w =$ size of element

$l =$ lower bound if not given take 0.

$$\begin{aligned} \text{loc } A[2] &= 1000 + 2(2-0) \\ &= 1000 + 4 \\ &= 1004 \end{aligned}$$

$$A[1 - 100] \quad b = 1000 \quad w = 4 \text{ bytes}$$

$$A[50] = 1000 + 4(50-1)$$

$$= 1196$$

Q. $A[-50 \dots 50]$ $b_a = 999$ $c_0 = 10$

↓ u

$$A[49] = 999 + 10 \times (49 - (-50))$$

$$= 1989$$

(13)

Two dimensional Array

2D array - An array which has 2 subscript is known as 1D array. It is also known as matrix.

row column
int a[2][3];

int a[2][3] = { {10, 20, 30}, {40, 50, 60} };

Memory representation: When 2D array get stored in computer memory. It stores in 1D way.

(1) Row major representation

(2) Column major representation

	a_{00}	a_{01}	a_{02}
0	10	20	3
1	40	50	60

10	20	3	40	50	60
----	----	---	----	----	----

10	40	20	50	3	60
----	----	----	----	---	----

$a[1 \dots 2, 1 \dots 3]$

$$\text{no of row} = \text{upper} - \text{lower} + 1 = 2 - 1 + 1 = 2$$

$$\text{no of column} = \text{upper} - \text{lower} + 1 = 3 - 1 + 1 = 3$$

$a[2 \dots 5, 7 \dots 12]$

$$\text{no of row} = 5 - 2 + 1 = 4$$

$$\text{no of column} = 12 - 7 + 1 = 6$$

$a[1 \dots 2, 1 \dots 3]$

	1	2	3
1	10	20	30
2	40	50	60

Row major : $[10 \ 20 \ 30 \ 40 \ 50 \ 60]$

Column major : $[10 \ 40 \ 20 \ 50 \ 30 \ 60]$

(14)

Row major and column major representation

Row major : $A[m][n]$

Address of $A[i][j] = b + [(i - l_y) * n + (j - l_c)] * c_2$

Column major

$$\text{Address of } A[i][j] = b + [(i - l_r) + (j - l_c) * m] * c_0$$

i = row of element to be found

j = column, , , , "

b → base address.

c_0 = size of element.

l_r = lower bound of row

l_c = lower bound of column

$$\text{Matrix } A[4][5] \quad ba = 1020 \quad c_0 = 2 \text{ byte}$$

$A[3][4]$

$$i = 3 \quad l_c = 0$$

$$j = 4 \quad l_r = 0$$

$$b = 1020 \quad m = 4$$

$$c_0 = 2 \text{ byte} \quad n = 5$$

$$R.M = b + [(i - l_r) * n + (j - l_c)] c_0$$

$$= 1020 + [(3 - 0) * 5 + (4 - 0)] 2$$

$$= 1020 + [15 + 4] * 2$$

$$= 1020 + 19 * 2 = 1058$$

$$= \underline{\underline{1058}}$$

(15)

More Examples on 2D

Q. $A[-15 \dots 10, 15 \dots 40]$ $ba = 1500$

$A[15 \dots 20]$

~~Ans~~

$$l_r = -15$$

$$u_r = 10$$

$$l_c = 15$$

$$u_c = 40$$

$$m = 10 + 15 + 1 = 26 \quad i = 15$$

$$n = 40 - 15 + 1 = 26 \quad j = 20$$

$$ba = 1500$$

$$\omega = 1 \text{ byte}$$

$$RM = 1500 + [(15 - (-15)) * 26 + (20 - 15)] * 1 \\ = 2285$$

(16)

3D Array

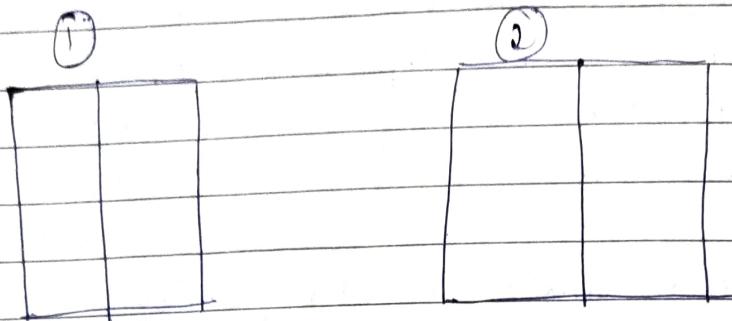
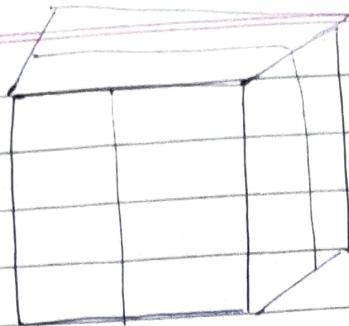
An array which has 3 subscript is known as 3D Array. 3D array is an array of 2D array.

datatype array name [Page] [Row] [Col]

int A [2] [5] [2].



5x2 ka 2 array hoga.



3-D Array Representation

- 1) Row major (C lang)
- 2) Column major (MATLAB)

(1)		(2)	
0	1		10
2	3		11
4	5		12
6	7		13
8	9		14
			15
			16
			17
			18
			19

Row major [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19]

Column major

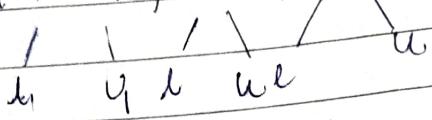
[0 | 10 | 2 | 12 | 4 | 14 | 6 | 16 | 8 | 18 | 1 | 11 | 3 | 13 | 5 | 15 | 7 | 17 | 9]

(7)

formula for Row major & column major

Consider 3D Array.

$$A[1 \dots p, 1 \dots R, 1 \dots C]$$



① Find length of all dimensions

$$L_1 = p - 1 + 1 \quad \text{Upperbound - lowerbound} + 1$$

$$L_2 = R - 1 + 1$$

$$L_3 = C - 1 + 1$$

② Find effective index for $A[k_1, k_2, k_3]$

$$E_i = k_i - \text{lower bound}$$

$$\text{Row major} = \text{Loc } A[k_1, k_2, k_3] = b + co((E_1 L_2 + E_2) L_3 + E_3)$$

$$\text{Column major} = \text{Loc } A[k_1, k_2, k_3] = b + co((E_3 L_2 + E_2) L_1 + E_1)$$

Row

$$E_1 * L_2 * L_3 +$$

Column

$$E_1 + E_2 + L_3$$

$$A[1 \dots 2, 1 \dots 5, 1 \dots 2]$$

$$k_1 = 1 \quad l_1 = 2 \quad E_1 = 0$$

$$k_2 = 3 \quad l_2 = 5 \quad E_2 = 2$$

$$b = 100 \quad co = 2 \quad k_3 = 1 \quad l_3 = 2 \quad E_3 = 0$$

$$\begin{aligned} RM \quad A[1, 3, 1] &= 100 + 2[(0 \times 5 + 2) \cdot 2 + 0] \\ &= 108 \end{aligned}$$

(19)

Multidimensional Array.

A n dimensional array has n subscript

$$A[1 \dots m_1][1 \dots m_2][1 \dots m_3] \dots [1 \dots m_n]$$

The element A with subscript denoted as

$$A[k_1, k_2, k_3, \dots, k_n]$$

The programming lang. will store array in

(1) Row major order.

$$\text{loc}(A[k_1, k_2, \dots, k_n]) = b + w[(E_1 L_2 + E_2) L_3 + E_3) L_4 + \dots + E_{n-1}) L_n + E_n]$$

(2) Column major order

$$\text{loc}(A[k_1, k_2, \dots, k_n]) = b + w[(L_1 E_n L_{n-1} + E_{n-1}) L_{n-2} + \dots + E_2) L_2 + E_1) L_1 + E_n]$$

(20)

Applications of Array.

- (1) Arrays are used to store list values.
- (2) Arrays are used to perform matrix operation.
- (3) Array are used to implement searching algorithm and sorting algorithm.
- (4) Array are used to implement stack and queue.
- (5) Array are used to represent sparse matrix.

(15)

Sparse Matrix

The situation in which a matrix contain more no. of zero element than non-zero element. Such matrix is called sparse matrix.

Advantage

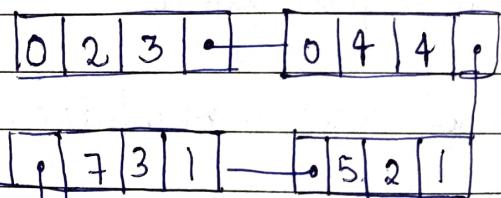
- 1) storage
- 2) computing time

Representation

(1) Array representn.
will represent in three field row, column & value

0	0	3	0	4
0	0	5	7	0
0	0	0	0	
0	0	2	6	0

linked list representn
will represent in 4 field
row, column, value, next node



row	0	0	1	1	3	3	3	1	2	0	3	2	6
col	2	4	2	3	1	2	7	3	1	5	2	1	8
val	3	4	5	7	2	6							

(Qd)

Operations on Linear Array.

- 1) Traversing
- 2) Insertion
- 3) Deletion
- 4) Sorting
- 5) Searching
- 6) Merging

1) Traversing

If we want to print element of linear array

LA:	10	20	12	13	15	
	↓ 0	1	2	3	4	↓ UB

LB

 $O(n)$

Algo

- 1) Set $K = LB$
- 2) Repeat Step 3 and 4 while $K \leq UB$.
- 3) Apply process to $LA[K]$
- 4) Set $K = K + 1$
- 5) exit

OR.

1. Repeat for $K = LB$ to UB .
2. Apply process to $LA[K]$
3. exit

```

#include <stdio.h>
void main()
{
    int k, LA[5] = {10, 20, 30, 15, 16};
    k = 0;
    while (k <= 4)
    {
        printf("%d", LA[k]);
        k++;
    }
}

```

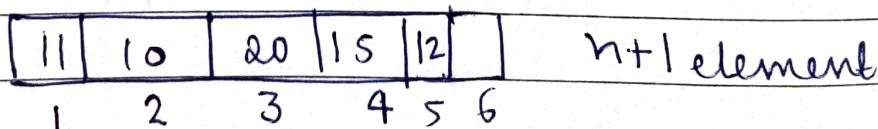
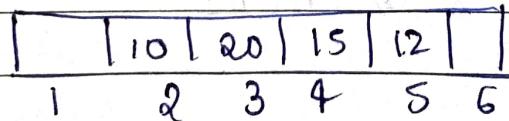
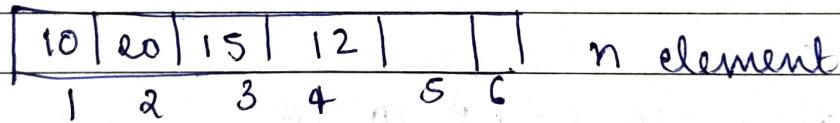
(23)

Insertion Operation

Insertion refers to opern. adding an element to linear Array

There are 3 cases

- ① Insert at begining O(n) worst case
- ② Insert at end O(1) best case
- ③ Insert at given loc. O(n) Avg case



At given loc.

10	20	15	12	
1	2	3	4	5 6



at 3

10	20		15	12	
1	2	3	4	5	6

10	20	11	15	12	
1	2	3	4	5	6

Algo

Insert (LA, N, K, item)

- 1) Set J=N
- 2) Repeat step 3 and 4 while $j \geq K$
- 3) Set $LA[j+1] = LA[j]$
- 4) Set $J = J - 1$
- 5) Set $LA[K] = item$
- 6) Set $N = N + 1$
- 7) exit

(24)

Deletion Operation.

Removing an element replacing it with next element

3 cases

- 1) Delⁿ from begining $O(n)$ worst
- 2) Delⁿ from end $O(1)$ best
- 3) Delⁿ from given locⁿ. avg case $O(n)$

30	40	25	27	35	$n=5$
1	2	3	4	5	

from
beg

40	25	27	35	$n-1 = 4 \text{ elements}$
1	2	3	4	5

worst case: $O(n)$

from
end

30	40	25	27	$O(1) \text{ best case}$
1	2	3	4	$n=4$

from given pos	30	40	25	27	35	$n-1 = 4 \text{ elements}$
	1	2	3	4	5	

$K = \text{pos}$

Algo.

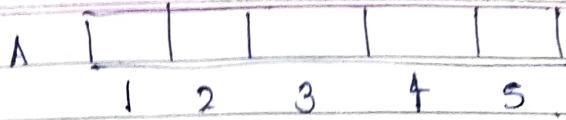
Delete (LA, N, K, item)

1. set item = LA[K]
2. Repeat from J = K to N-1
3. Set LA[J] = LA[J+1]
4. Set N = N - 1
5. Exit

(25) Sorting

Bubble Sort Algorithm

Sorting refers to the operation of rearranging elements of arrays in increasing order.



$$A[1] < A[2] < A[3] < A[4] < A[5]$$

Bubble sort - simplest way to sort an array

Assume $A[1], A[2], \dots, A[n]$

- First compare $a[1]$ and $a[2]$ if $a[2]$ is less than $a[1]$ swap $a[1]$ and $a[2]$. $a[1] < a[2]$

Pass 1 { Compare $A[1]$ and $A[2]$ and arrange in order $A[1] < A[2]$
 { Compare $A[2]$, , $A[3]$ $A[2] < A[3]$
 \vdots
 $(n-1)$ { " , , $A[N-i]$, , $A[N]$. $A[N-i] < A[N]$

Pass 2 $(n-2)$

Pass 3 $(n-3)$

Pass $N-1$ Compare $A[1]$ and $A[2]$ and arrange $A[1] < A[2]$

<u>20</u> 19 <u>13</u> 25 15	Pass 2	<u>13</u> 19 20 15 25
1 2 3 4 5 .		1 2 3 .

Pass 1	<u>19</u> 20 13 25 15
4 comp	1 2 .

$n-2$ comparisons.

<u>19</u> 13 20 25 15
1 2 3 .

<u>19</u> 13 20 15 25
1 2 3 4 .

Pass 3

13 | 19 | 15 | 20 | 25

 └┘

13 | 19 | 15 | 20 | 25

 └┘

13 | 15 | 19 | 20 | 25

Pass 4

13 | 15 | 19 | 20 | 25

 └┘

DATA

13 | 15 | 19 | 20 | 25

$$n-1 + n-2 + n-3 + \dots + 1 = \frac{n(n-1)}{2}$$

$$\frac{n^2 - n}{2}$$

$$T(n) = O(n^2)$$

Algo

BUBBLE (DATA, N)

1. Repeat step 2 and 3 for k=1 to N-1
2. Set PTR=1
3. Repeat while PTR < N-k
 4. if DATA[PTR] > DATA[PTR+1]
 5. swap DATA[PTR] and DATA[PTR+1]
 6. Set PTR= PTR+1
7. exit

(26)

Searching techniques

Linear search

Searching algo are designed to check an element or retrieve element from an array.

Generally searching is classified in 2 categories

A	10	20	15	18	29	item = 15
	0	1	2	3	4	
	↑	↑	↑			item = 30

Algo

- Search (A, N, ITEM, LOC)

1. Repeat Step 2 for $i=0$ to $N-1$
2. if ($A[i] == item$)
 - LOC = i ;
 - break;
3. if ($i == n$)
 - Print "Element not found"
- else
 - Print "Element found at LOC"
4. Exit

Best case: O(1)
Worst case: O(n)
Avg case: O(n)

Date: / /
Page No.

```
#include<stdio.h>
void main()
{
    int a[5] = {10, 20, 15, 18, 29};
    int i, item, loc;
    printf("Enter item to be searched")
    scanf("%d", &item)

    for(i=0; i<5; i++)
    {
        if(a[i] == item)
        {
            loc = i;
            break;
        }
    }

    if(i == 5)
        printf("Item not found")
    else
        printf("Item found at %d", loc);
```

(27)

Binary Search.

Binary search is a technique which works on sorted Array. It works on divide & conquer approach.

Limitations

- i) As a input we need to give sorted array.

Algo.

Binary search(A, LB, UB, ITEM, LOC)

1. BEGIN = LB END = UB
2. MID = ((LB + UB) / 2)
3. Repeat step 3 and 4 while BEGIN < END and A[MID] ≠ item
4. if Item < A[MID] then
 set END = MID - 1
 else
 set BEGIN = MID + 1
5. set MID = ((BEGIN + END) / 2)
6. if A[MID] = ITEM then
 set LOC = MID
 else
 set LOC = NIL
7. exit

```
#include <stdio.h>
void main()
{
    int i, n, beg, end, mid, item, a[100];
    scanf("%d", &n);
    printf("Enter elements for array");
    for (int i = 0; i < n; i++)
        printf(" %d", &a[i]);
    printf("Enter the key");
    scanf("%d", &item);
    beg = 0;
    end = n - 1;
    mid = ((beg + end) / 2);
```

```

while ((beg <= end) && a[mid] != item)
{
    if (item < a[mid])
        end = mid - 1;
    else
        beg = mid + 1;
    mid = (beg + end) / 2
}
if (a[mid] == item) {
    printf ("item found at %d", mid);
} else {
    printf ("item not found");
}

```

Best case $O(1)$ $a[mid] = item$

Avg case $O(\log n)$

Worst case $O(\log n)$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$x = \log_2 n$$

no of element in array = 16.

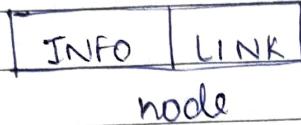
$\log_2 16 = 4$ comparisons gives 9%.

(27)

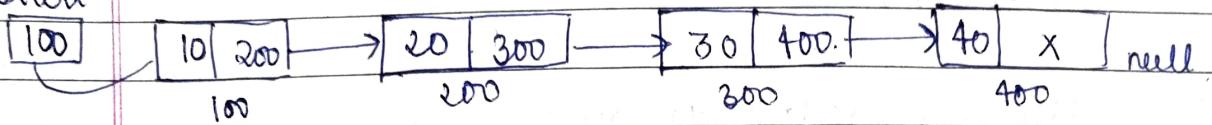
linked list

linked list or one way list is a linear collection of data elements called node where linear order given by pointer.

each node divided into 2 part



Start



Advantage

- 1) Dynamic in size
- 2) Ease of insertion and deletion

Disadvantage

- 1) Random access not allowed. (Binary search x)
- 2) Extra memory used at every node.

(2B)

Pointee Implementation of linked list

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void create();
void display();
  
```

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

```
struct node *start = NULL;
```

```
int main() {  
    int choice;  
    while(1);  
}
```

```
    printf ("1. Create \n");  
    printf ("2. Display \n");  
    printf ("3. Exit \n");
```

```
    printf ("Enter your choice")  
    scanf ("%d", &choice);  
    switch (choice)
```

```
    case 1: create();  
            break;
```

```
    case 2: display();  
            break;
```

```
    case 3: exit(0);  
            break;
```

```
    default: printf ("Wrong choice");  
}
```

```
return 0;
```

```
}
```

void create ()

{

```
struct node *temp, *ptr;
temp = (struct node *) malloc ( sizeof ( struct node ) )
scanf ("%d", &temp->info);
temp->next = NULL;
```

```
if ( start == NULL ) {
```

```
    start = temp;
```

}

```
else {
```

```
    ptr = start;
```

```
    while ( ptr->next != NULL )
```

```
        ptr = ptr->next
```

```
    ptr->next = temp;
```

}

}

void display ()

{

```
struct node *ptr;
```

```
printf ("\n list of elements are ");
```

```
for ( ptr = start ; ptr != null ; ptr = ptr->next )
```

```
    printf ("%d", ptr->info);
```

}

(30)

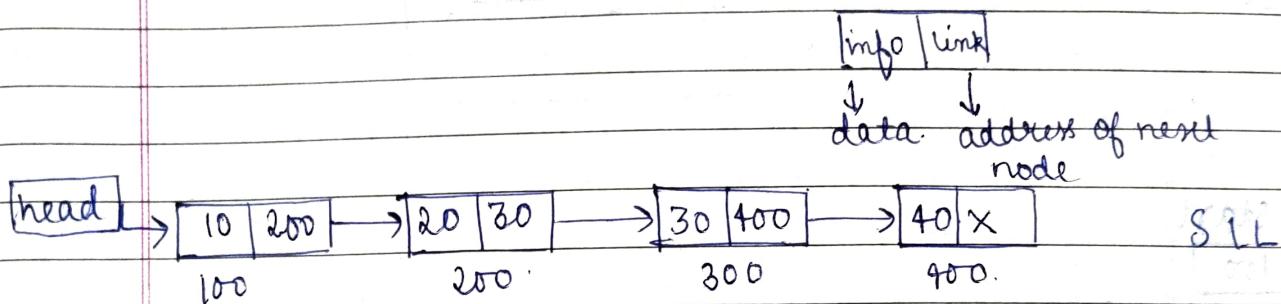
Differences between linked list and Array.

		dinked list
1) Array	1) Array is a collection of similar data.	1) dinked list is an ordered collection of same type, each element connected using pointer.
2) Array element can be accessed randomly using array index.	2) Random Access not possible. Element can be accessed sequentially.	
3) Data elements are stored in contiguous location in memory.	3) New elements can store anywhere and reference is created using pointer.	
4) Insertion, deletion is not easy.	4) "Insertion & Del" is easy in LL.	
5) Memory allocation during compile time (Static memory allocation)	5) Memory allocation during run time (dynamical memory allocation)	
6) Size of array must be specified at time of declaration.	6) Size of linked list shrink and grows when a new element deleted/inserted.	
7) A: [10 20 30 40 50 60 70]		<pre> graph LR Start((Start)) --> Node1[10] Node1 --> Node2[20] Node2 --> Node3[30] Node3 --> Node4[40] Node4 --> Node5[50] Node5 --> Node6[60] Node6 --> Node7[70] Node7 --> Null(()) </pre>

(31)

Types of linked list

- 1) Singly LL (one way)
- 2) Doubly LL (2 way)
- 3) Circular LL



DLL

Start
100

Frontwards .

add of new node

↑
[BACK INFO FORWARD]

↓
data
address of
next
node

Struct node {

int info ;

Struct node * forward;

Struct node * back;

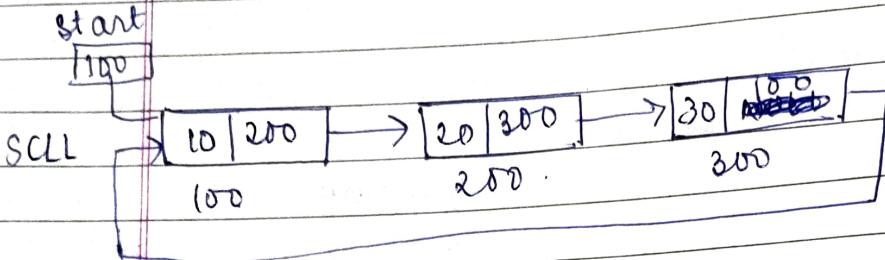
}

, backwards .

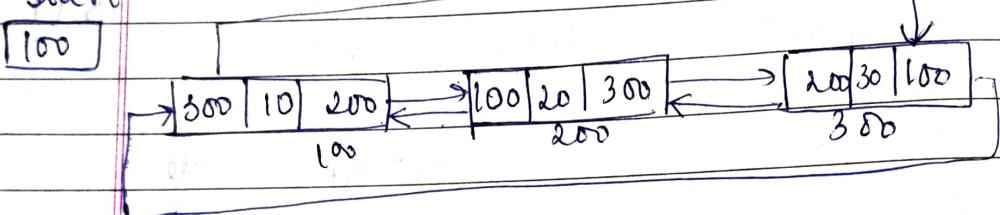
CLL

singly CLL

doubly CLL



Start



(32)

Operations on LL.

- 1) Traversal
- 2) Insertion
- 3) Deletion

Traversal

- 1) Start with head of first & access data
- 2) Go to the next node and access data
- 3) Continue until last node.

Algo

1. Set PTR = START
2. Repeat Step 3 and 4 until PTR != NULL
3. Write INFO(PTR)
4. PTR = LINK(PTR)
5. exit

C-program

```
Struct node{
```

```
    int data;
```

```
    Struct node * next;
```

```
}
```

```
Struct node * temp = head;
print ("list of elements");
while (temp != NULL) {
    print ("%.d", temp->data);
    temp = temp->next;
}
```

Time complexity : O(n)

(33)

Insertion in a LL

- a) Add "n" in begining
- b) Add "n" at end
- c) Add to the middle.

Add to beginning

- 1) Allocate memory to new node
- 2) store data
- 3) change next of new node to point to head
- 4) change head to point recently created node

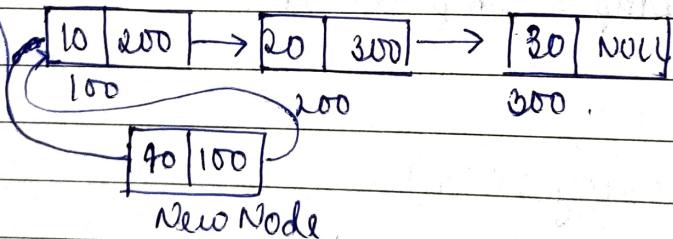
C program

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

```
struct node * newNode;  
newNode = malloc( size of ( struct Node ) );  
newNode → data = 40;  
newNode → next = NULL head;  
head = newNode
```

head

400



Add at the end.

- 1) Allocate memory to new node.
- 2) store data
- 3) Traverse to last node.
- 4) Change next of last node to recently created node.

C program:

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

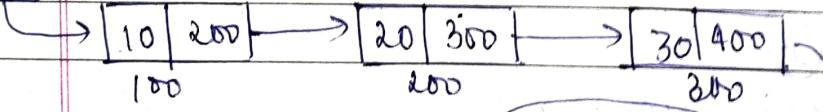
```
struct node* newNode;
newNode = malloc (size of (struct node));
newNode->data = 40;
newNode->next = NULL;
```

~~addition~~

```
struct node *temp = head;
while (temp->next != NULL)
    temp = temp->next;
temp->next = newNode;
```

head

100



temp

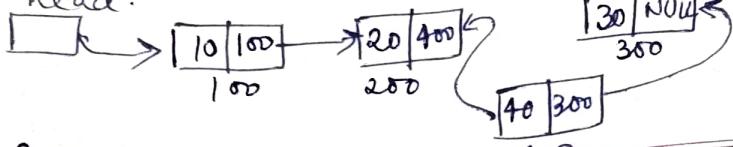
100

→ [40 | NULL]

400

Add " at middle

- 1) Allocate memory to the new node.
- 2) Store data
- 3) Traverse to the node just before pos.
- 4) Change the pointer to include new node in between.



C program

struct node {

int * data;

struct node* next;

};

struct node* newNode;

Newnode = malloc (sizeof(struct node));

newnode → data = 40;

int pos, i;

printf ("Enter position")

scanf ("%d", &pos);

struct node* temp = head;

for (i=2; i<pos; i++) {

if (temp → next != NULL)

temp = temp → next;

}

new node → next = temp → next;

temp → next = newnode;

(34)

Deletion from a LL.

a) From beginning

Point head → second node

head = head → next

b) From end

i) Traverse to second last element

ii) Change its next pointer to null.

```

struct node *temp = head;
while ((temp->next->next != null))
    temp = temp->next;
temp->next = null;

```

c) from middle or pos.

- Debounce to element before the element to be deleted
- Change the next pointer

```
for (i = 2; i < position; i++)
{
```

```
if (temp->next != null)
```

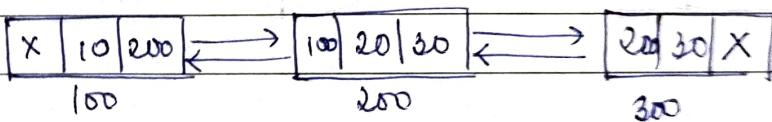
```
temp = temp->next
```

```
temp->next = temp->next->next
```

(35)

Doubly linked list
(two way list)

prev	data	next
------	------	------



```

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

```

Operations on DLL

- Insertion
- deletion
- Traversal

Memory representation.

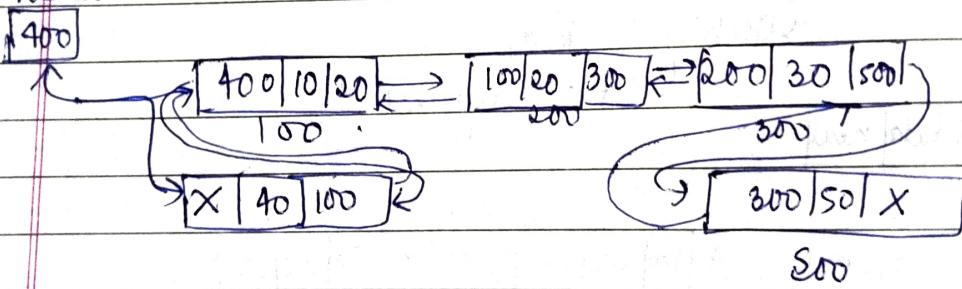
Start	Data	Prev	Next
100	10	NULL	200
200			
300	20	100	500
400			
500	30	200	600
600	40	500	NULL

forward (next)

1) Traversal ↙ backward (prev)

2) Insertion ↙ beg
 end
 ↘ at locⁿ.

head

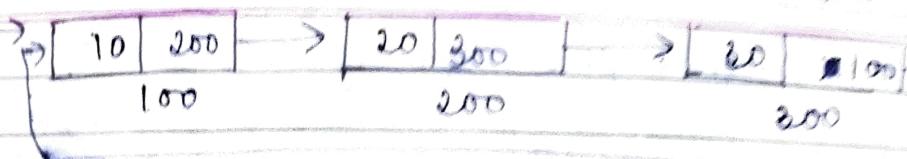


3) deletion ↙ beg
 end
 ↘ at locⁿ.

(3b)

Circular linked list

head
100

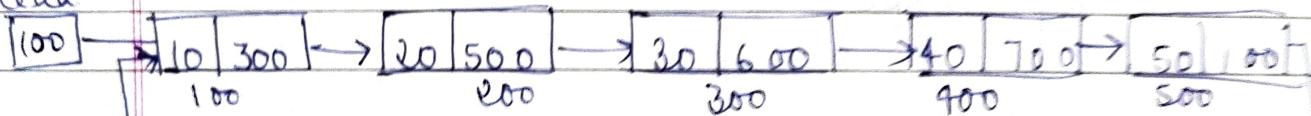


23
Date: _____
Page No. _____

Memory representation

	data	next
100	10	200
200		
300	20	500
400		
500	30	600
600	40	700
700	50	100

head



Operations

1) Traversal

2) Insertion

3) Deletion



a) beg

b) end

c) At pos.

a) beg

b) end

c) at pos.

(34)

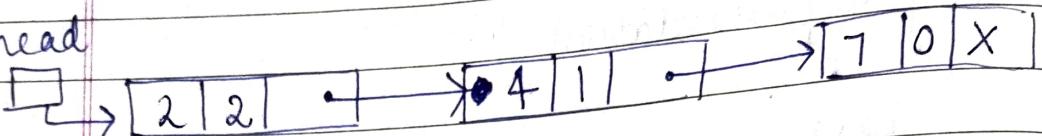
Polynomial representation using LL

Linked list is used to represent polynomial of any degree. Polynomial consists of variable with coefficient and exponent.

coeff | expo | link

$$2x^2 + 4x + 7x^0.$$

head



struct node {

int coeff;

int expo;

struct node* next;

};

$$5x^4 + 7x^2 + 32x^0.$$

coeff expo .

5 4

7 2

32 0.



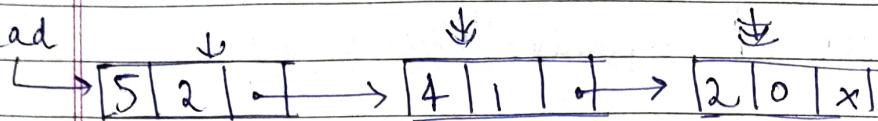
Addition of polynomial

- 1) Loop around all values of linked list
- 2) If value of node exponent is greater copy this node to result and head point it.
- 3) If the value of both exp is same add coeff and add to result.
- 4) Print result.

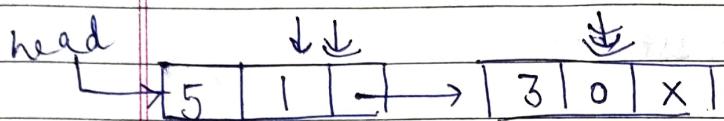
$$p(1) : 5x^2 + 4x + 2$$

$$p(2) \quad \begin{matrix} 5x+3 \\ 5x^2 + 9x + 5 \end{matrix}$$

head



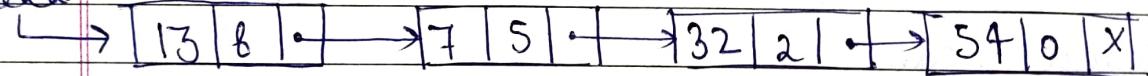
head



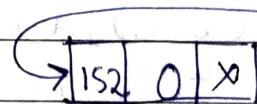
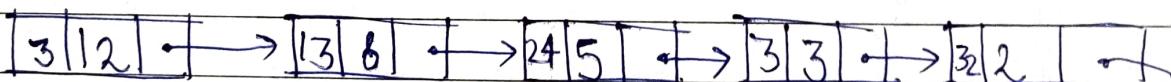
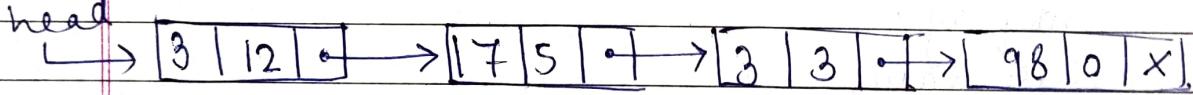
$$p(1) : 13x^8 + 7x^5 + 32x^2 + 54$$

$$p(2) : 3x^{12} + 17x^5 + 3x^3 + 98.$$

head



head



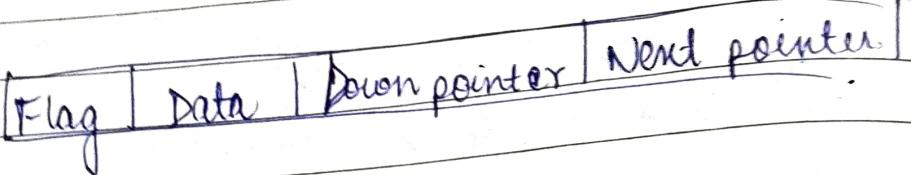
(38)

Generalized LL.

A Generalized linked list is defined as a sequence of $n \geq 0$ elements $l_1, l_2, l_3, \dots, l_n$ such that l_i are either atom or list of atoms.

Page No. _____

$L = (l_1, l_2, l_3, \dots, l_n)$ where n is
total no of atom.



Flag : 0 \rightarrow next pointer exist
1 \rightarrow down pointer "

Data : atom

Down p : address to down node

Next p : address to next node

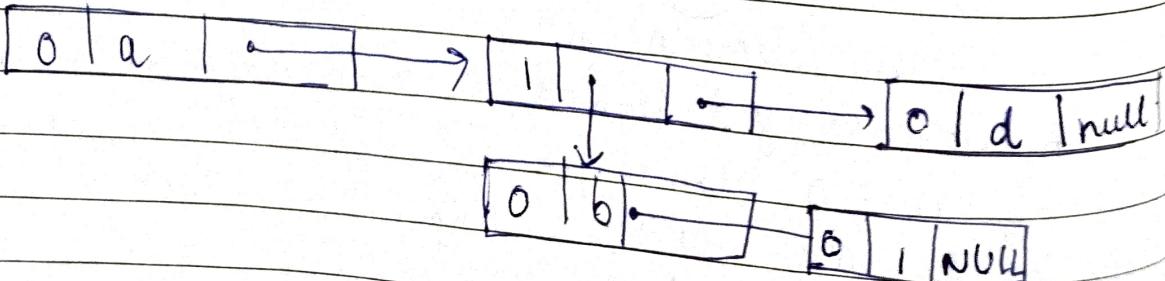
Struct node

```
int flag;  
char data;
```

```
Struct node * down, * next;  
};
```

1) $(a, (b, c), d) = L$

|
list of
atoms.
|
① ② ③



b) $(p, q, r, s(t \cup v), w) x, y)$

① ②

③

④ ⑤

$0 | p \rightarrow 0 | q \rightarrow 1 | \downarrow \rightarrow 0 | n : \rightarrow 0 | y | x$

$0 | r \rightarrow 0 | s \rightarrow 1 | \downarrow \rightarrow 0 | w | x$

$0 | t \rightarrow 0 | u \rightarrow 0 | v | x$

③

Multivariable Polynomial.

We used generalized LL to represent multivariable poly.

$$9x^5 + 7x^4y + 10xz$$

flag	data	down p	next p
------	------	-----------	-----------

Flag: 0: Variable present

1: down variable present

2: coeff exp. present.

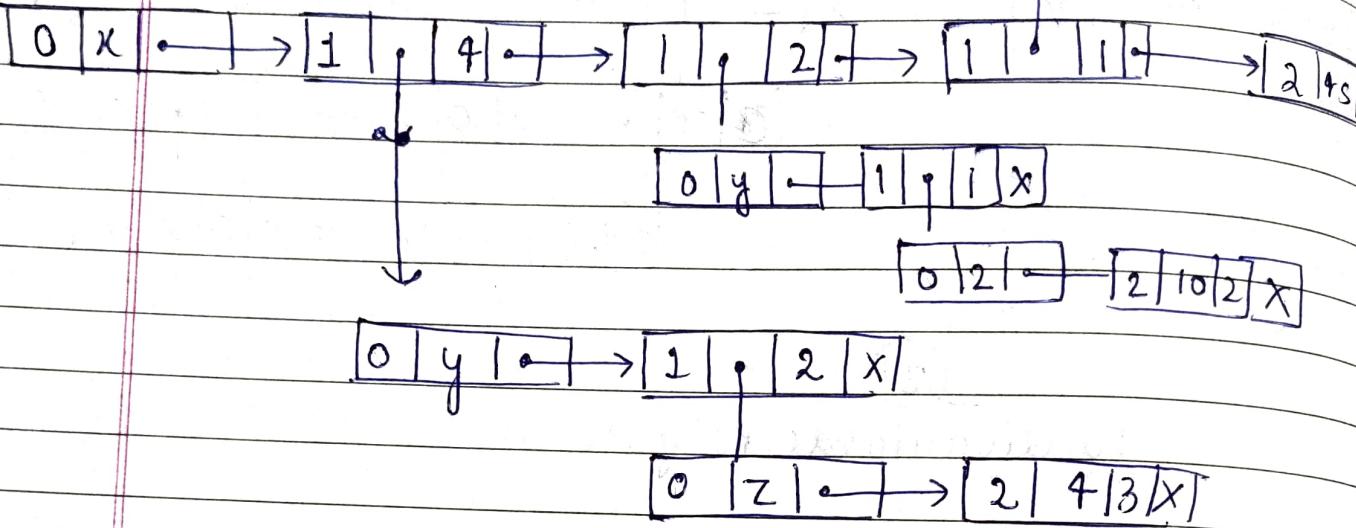
$0 | x \rightarrow 2 | 9 | 5 \rightarrow 1 | \downarrow 4 \rightarrow 1 | \downarrow 1 | n | u$

$0 | z \rightarrow 2 | 1 | 0 | l | x$

$0 | y \rightarrow 2 | 7 | 1 | x$

$$Q. -4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45.2yz^0.$$

~~(Q. 8)~~



⑩

Stack in Datastructure

A stack is a list of elements in which an element may be inserted or deleted only at one end called "TOP" of stack. Stack is sometime called LIFO or FILO.

Eg → stack of plates
stack of books.

Features of stack

- 1) Stack is an ordered list of similar datatype
- 2) Stack is LIFO or FILO structure
- 3) Push() or Pop() functions used
- 4) Stack is overflow - ~~full~~ full underflow - empty

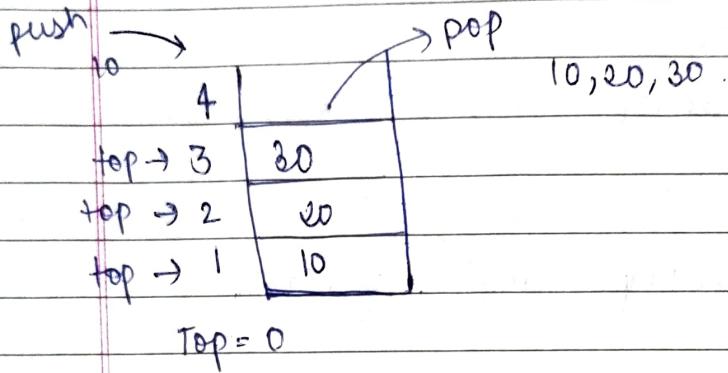
$$2 \frac{1}{3} \times \frac{4}{5} \text{ of } \frac{45}{7} + 3 \frac{1}{2} \div 6 \frac{1}{8}$$

$$\frac{11}{3} \times \frac{4}{5}$$

Date _____ / _____ / _____
Page No. _____

Applications

- 1) Recursion
- 2) Expression evaluation (infix, prefix, postfix)
- 3) Parsing
- 4) Tree traversal.
- 5) Backtracking.

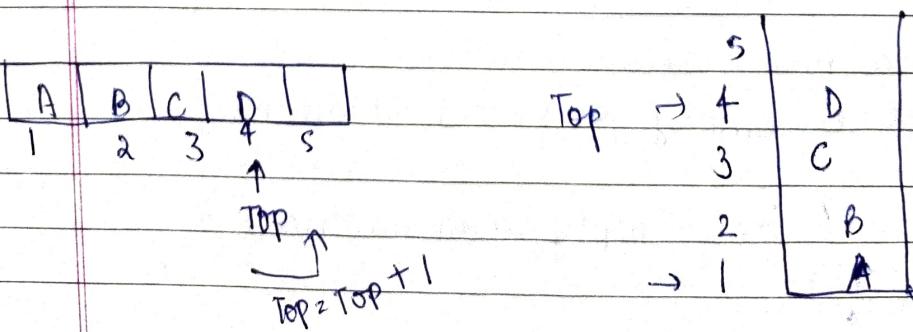


(41)

Array implementation of Stack.

Operations of Stack

- 1) Push → insertion
- 2) Pop → deletion
- 3) IsEmpty → is stack empty
- 4) IsFull → is stack full
- 5) Peek → Top position (displays the value)



Push()

1. if top = n (overflow)
2. top = top + 1
3. stack [top] = item
4. Exit

Pop()

1. if Top = 0 (underflow)
2. item = stack [top]
3. Top = Top - 1
4. end

Top = 0 (Is empty)

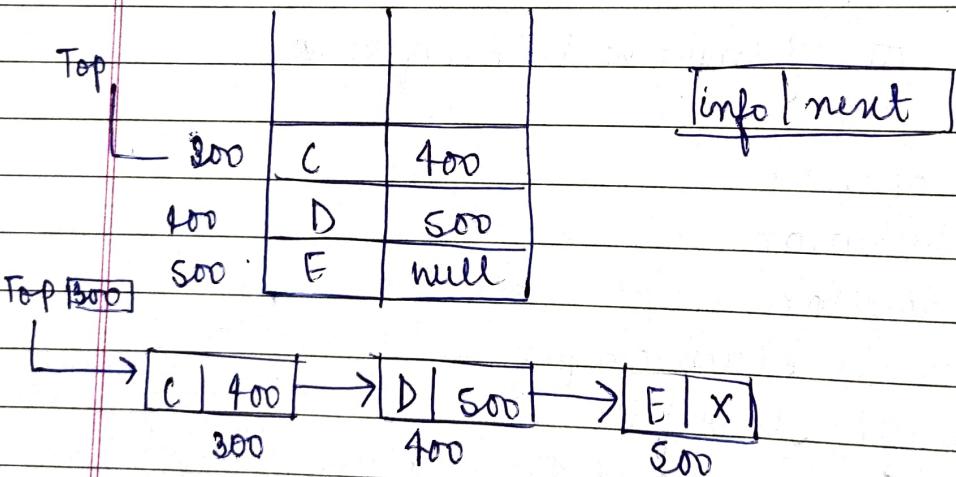
Top = n (Is full)

peek = D (value at top)

(42)

linked list implementation of stack

linked list allocate memory dynamically



Push

1. Create a new node
2. If stack is empty push as start node.
3. If list is not empty add new node to start of list.

Pop

1. check underflow condition
2. adjust head pointer (top)

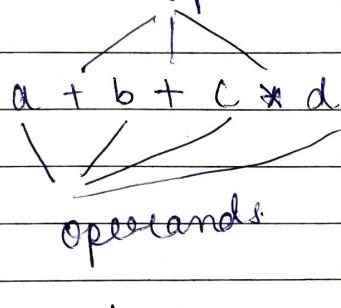
(43)

Arithmetic Expressions

Polish notation

Stack applications

1. Arithmetic Expressions : AE involves operands & operators



1. Infix expression
2. Prefix expression (polish notation)
3. Postfix expression (reverse polish notation)
()

Highest : \uparrow (exponent)

Next highest : $*$ /

Lowest : +, -

$() \uparrow * / > +, -$

$$2^3 + 5 * 2^2 - 12 / 6$$

$$8 + 5 * 4 - 12 / 6$$

$$8 + 20 - 2$$

$$= 26$$

It needs parentheses & operator precedence.

Infix: operand1 operator operand2 ($A + B$)

Postfix: operand1 operand2 operator ($AB +$)

Prefix: operator operand1 operand2 ($+ AB$)

(4)

Infix \rightarrow Postfix conversion

Arithmetic expression

1. Infix to Postfix
2. Infix to Prefix
3. Postfix to Infix
4. Postfix to Prefix
5. Prefix to Infix
6. Prefix to Postfix

Postfix (Q, P)

1. Push 'l' onto stack and add 'l' to end of P.
2. Scan Q from left to right and repeat step 3 to 6.
3. If an operand encountered add to P.
4. If left parenthesis encountered push to stack.
5. If operator (+) is encountered then
 - a) Repeatedly pop from stack add to P each operator same or higher precedence.
 - b) Add (+) to stack.
6. If right parenthesis encountered then
 - a) Pop from stack & add to P until left parenthesis.
 - b) Remove left parenthesis.
7. Exit.

$$Q. \quad A + (B * C - (D / F \uparrow F) * G) * H$$

Oziva

SPS

Date: / /

Page No.

Symbol	Stack	Postfix expression
((
A	(A
+	(+	
((+(
B	(+()	AB
*	(+(*)	AB.
C	(+(*)	A B C
-	(+ (-	A B C *
((+ (- (A B C *
D	(+ (- (A B C * D.
/	(+ (- (/	A B C * D.
E	(+ (- (/	A B C * D E
↑	(+ (- (/ ↑	A B C * D E
F	(+ (- (/ ↑	A B C * D E F
)	(+ (-	A B C * D E F ↑ /
*	(+ (- *	A B C * D E F ↑ /
G	(+ (- *	A B C * D E F ↑ / G
)	(+ *	A B C * D E F ↑ / G * -
*	(+ *	A B C * D E F ↑ / G * -
H	(+ *	A B C * D E F ↑ / G * - H
)		A B C * D E F ↑ / G * - H * +

45

Examples on Infix to Postfix

Q. 1) $A * (B + D) / E - F * (G + H / K)$

Postfix Express

Symbol	Stack	Postfix Express
((
A	(A
*	(*	A.
B	(*	AB.
+	(+	AB*
D	(+	AB*D.
)		AB*D+
+		
C	(* (A
B	(* (AB.
+	(* (+	AB.
D	(* (+	ABD.
)	(*	ABD+
/	(* /	ABD+*
E	(/	ABD+* E
-	(-	ABD+* E /
F	(-	ABD+* E / F
((- (ABD+* E / F
G	(- (ABD+* E / F G
+	(- (+	ABD+* E / F G
H	(- (+	ABD+* E / F G H
/	(- (+ /	ABD+* E / F G H
K	(- (+ / K	ABD+* E / F G H K
)	(-	ABD+* E / F G H K /
)		ABD+* E / F G H K / + -

46

Infix to Prefix

- 1) Reverse the infix expression
- 2) Apply infix to postfix algorithm to obtain postfix
- 3) Reverse the postfix expression to obtain prefix.

Ex: $(d-c) * (b-a)$

$(a-b) * (c-d)$

Symbol	Stack	Postfix
(
((
a	(a
-	(-	a
b	(-	ab
)	(ab -
*	{ *	ab -
({ * (ab -
c	{ * (ab - c
-	{ * (-	ab - c
d	{ * (-	ab - cd
)	{ * ()	ab - cd -
		ab - cd - *

Prefix: $(* - dc - ba)$

(47)

Postfix to Infix

1. Read the postfix expression from left to right.
2. If we read operand push to stack
3. If we read operator pop top two
4. go to step 1 until completed.
 first operand $\rightarrow O_{P2}$
 second operand $\rightarrow O_{P1}$

Ex: ab + cd - * O_{P1} operator O_{P2}

$O_{P2} \rightarrow$	a	b	$a+b$
$O_{P1} \rightarrow$			

$-$	d	c	$c-d$
		$a+b$	$a+b$

$*$	$c-d$	$(a+b) * (c-d)$
$O_{P2} \rightarrow$	$a+b$	
$O_{P1} \rightarrow$		

Postfix to prefix

- 1) Read the postfix expression from left to right
- 2) If we read operand push to stack
- 3) If we read operator POP Top two operand
 - a) first operand called OP2
 - b) second operand called OP1
 - c) make expression (operator OP1 OP2)
- 4) goto step 1 until complete.

		+	
OP ₂ →		b	+ ab .
OP ₁ →		a	

		-	
OP ₂ →		d	- cd .
OP ₁ →		c	
		+ ab	

		*	
OP ₂ →		- cd	* + ab - cd .
OP ₁ →		+ ab	

48

Prefix to Infix & postfix

Prefix to Infix

1. Reverse the prefix expression
2. Read the expression left to right
3. If we read operand push to stack
4. If we read operator POP top two operand
 - a) First operand called OP1
 - b) Second operand called OP2
 - c) make expression (OP1 operator OP2)
 - d) Put this expression to stack
- 5) go to step 1 until complete.

In: $* + ab - cd$

Prefix: $* + ab - cd$.

Reverse: dc - ba + *

		-	
OP2 \rightarrow		c	. c - d
OP1 \rightarrow		d	

		+	
OP1 \rightarrow		a	a + b
OP2 \rightarrow		b	

		*	
OP1 \rightarrow		a + b	(a + b) * (c - d)
OP2 \rightarrow		c - d	

Prefix to Postfix

- 1) Reverse the prefix expression
- 2) Read the expression left to right
- 3) If we read operand push to stack
- 4) If we read operator POP Top two operand
 - a) First operand called OP1
 - b) Second operand called OP2
 - c) make expression OP1 OP2 operator
 - d) Put this to stack
- 5) Go to step 1 till complete

Prefix: $* + ab - cd$

Reverse: $dc - ba + *$

Postfix: $ab + cd - *$

$\text{oper} \rightarrow$	-	$cd -$
$OP_1 \rightarrow$	c	
$OP_2 \rightarrow$	d	

$OP_1 \rightarrow$	+	$ab +$
$OP_2 \rightarrow$	a	
	b	
	$cd -$	

*		
ab +		$ab + cd - *$
cd -		

Prefix to Postfix

- 1) Reverse the prefix expression
- 2) Read the expression left to right
- 3) If we read operand push to stack
- 4) If we read operator POP Top two operand
 - a) First operand called OP1
 - b) Second operand called OP2
 - c) make expression OP1 OP2 operator
 - d) Put this to stack
- 5) Go to step 1 till complete

Prefix : * + ab - cd

Reverse : dc - ba *

Postfix : ab + cd - *

$\text{oper} \rightarrow$	-	cd-
$OP_1 \rightarrow$	c	
$OP_2 \rightarrow$	d	

$OP_1 \rightarrow$	+	ab+
$OP_2 \rightarrow$	a	
	b	
	cd-	

*	
ab+	ab + cd - *
cd-	

49

Evaluation of Postfix Expression

Algorithm.

end

- 1) Add ')' at the ~~start~~ of expression.
- 2) Scan expression from left to right until ')' encountered.
- 3) If an operand encountered push to stack.
- 4) If an operator (+) encountered then
 - a) POP top 2 operand from stack
 - b) first POP operand is OP₁
Second POP operand is OP₂.
 - c) evaluate OP₂ \oplus OP₁
 - d) Push to stack.
- 5) Top of stack is final value.
- 6) Exit

P: 5 6 2 + * 12 4 / -)

Symbol	Stack	
5	5	OP ₁ S
6	5, 6	OP ₂ S
2	5, 6, 2	
+	5, 8	6 + 2 = 8
*	40	5 * 8 = 40
12	40, 12	
4	40, 12, 4	
-	40, 3	12 / 4 = 3.
)	37	40, 3 = 3

56

Recursion Implementation in Stack

The process in which a function call itself directly or indirectly is called Recursion. In recursion a function 'A' either call itself directly or call another function 'B' that is called function.

`fun()`

`fun1()`

}

`fun();` → direct recursion

`fun2()`

`fun2()`

`fun1();`

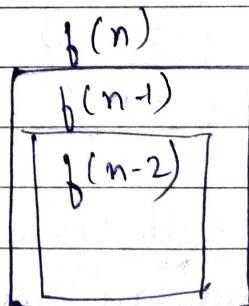
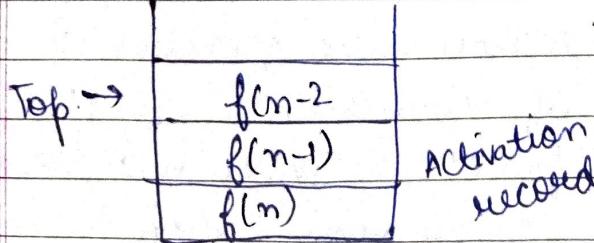
}

Indirect recursion

Properties of recursion

1. Base criteria
2. Progressive approach

Stack implementation

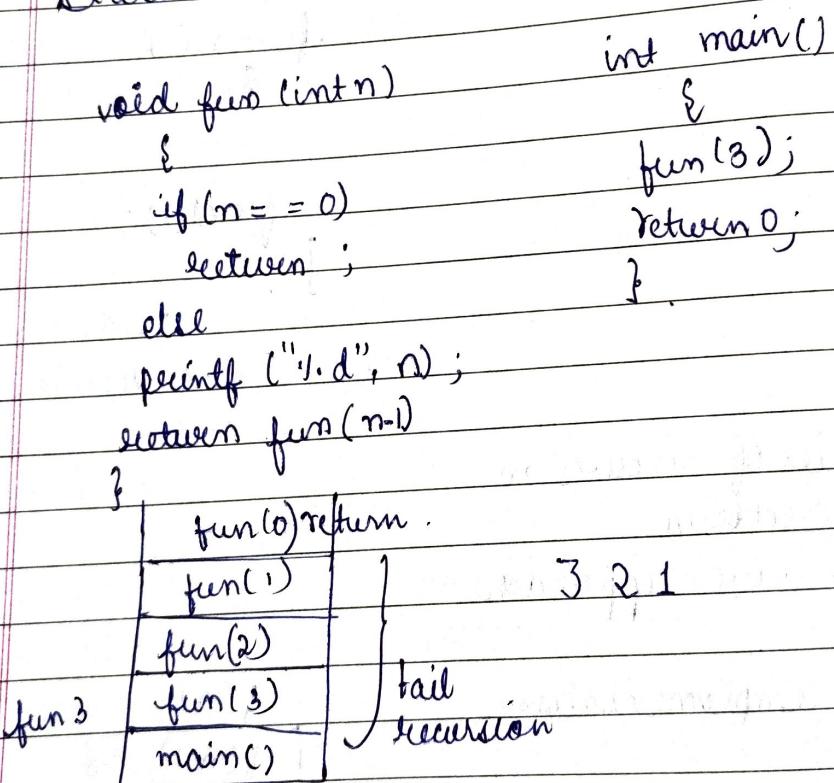


(51)

Types of Recursion

- 1) Direct recursion
- 2) Indirect recursion
- 3) Tail recursion
- 4) Non tail recursion.

Tail recursion: A recursive function is called tail recursive if recursion is the last thing done by function. There is no need to keep record of previous state.



Non tail recursion: A recursive function is called non tail recursive if recursion is not the last thing done by funcn. There is a need to keep record of previous stack.

```

void fun(int n)
{
    if (n == 0)
        return;
    else
        fun(n - 1);
    printf ("%d", n);
}

```

```

int main()
{
    fun(3);
    return 0;
}

```

%P → 1 2 3

(52)

Recursion Algorithm for factorial.

The product of the no's from 1 to n is called factorial of n denoted by $n!$.

$$n! = 1 * 2 * 3 * \dots * n$$

$$n! = n * (n-1) * (n-2) * \dots * 1$$

$$1! = 1 \quad 0! = 1$$

$$2! = 1 * 2 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

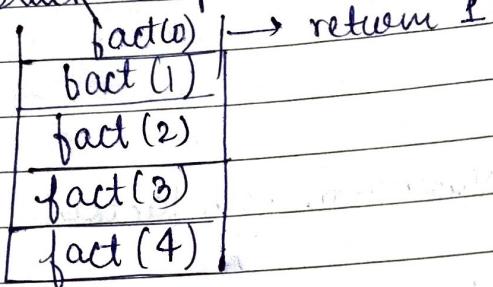
$$\text{fact}(n) = \begin{cases} 1 & n = 0 \\ n * \text{fact}(n-1) & n > 0 \end{cases}$$

int fact (int n)

{
if ($n = 0$)
return 1 ;

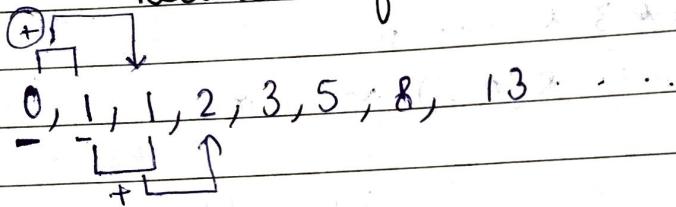
else
return $n * \text{fact}(n-1)$;
}

Stack Implementation.



(53)

recursion fibonacci series



0	1	1	2	3	5	8	13
0	1	2	3	4	5	6	7

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = 1 + 1 = 2$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1 + 0 = 1$$

$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \rightarrow$ general formula.

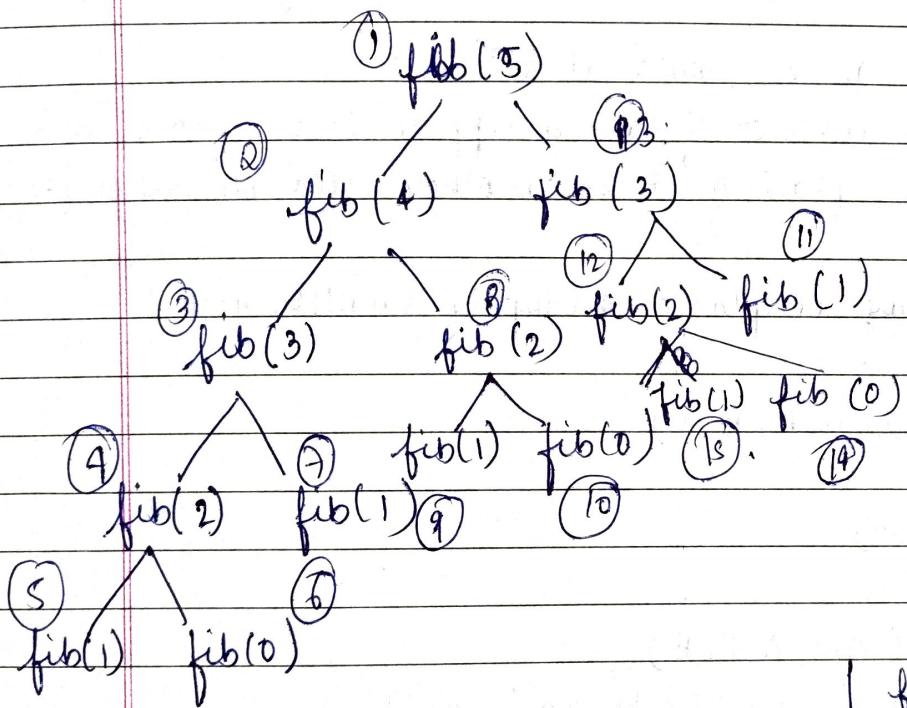
$$\text{fib}(n) = \begin{cases} n & n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & n > 1 \end{cases}$$

Recursive funcⁿ.

```

int fib(n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}

```



$\text{fib}(0) = 0$
$\text{fib}(1) = 1$
$\text{fib}(2) =$
$\text{fib}(3) =$
$\text{fib}(4) =$
$\text{fib}(5) =$

(54)

Tower of Hanoi

Revision

- 1) Factorial
- 2) Fibonacci
- 3) Tower of Hanoi

Tower of Hanoi is a mathematical puzzle invented by mathematician Lucas in 1883. In this puzzle we have 3 ~~wooden~~ rods and n disk objective puzzle to move entire disk from first rod to another by following

1. One disk can be moved at a time.
2. Each move consist of taking upper disk from one of rod and placing it on another rod or an empty rod.
3. No disk may be placed on top of smaller disk.

```
void TOH ( n, A, B, C )
```

{

```
if (n > 0)
```

{

```
TOH ( n-1, A, C, B )
```

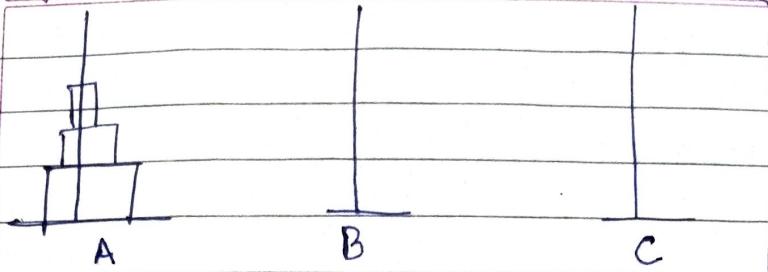
```
print (' from ' + d + ' to ' + d', A, C );
```

```
TOH ( n-1, B, A, C )
```

}

$n(A, B, C)$ using
from to

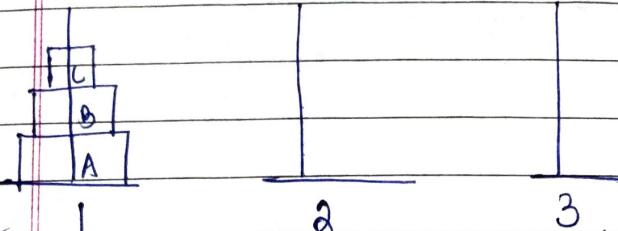
SPS
Date: / /
Page No.



Taking an example of 3 disk

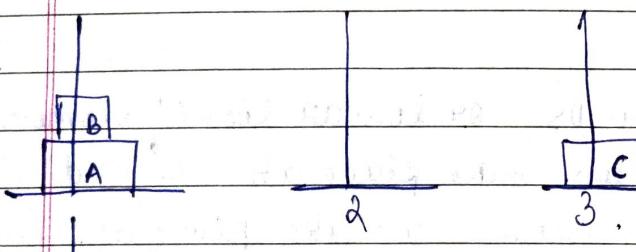
next

$1 \rightarrow 3$



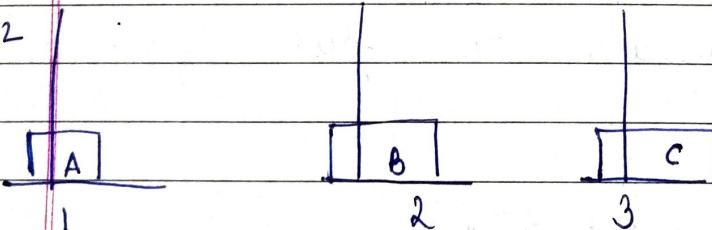
next

$1 \rightarrow 2$



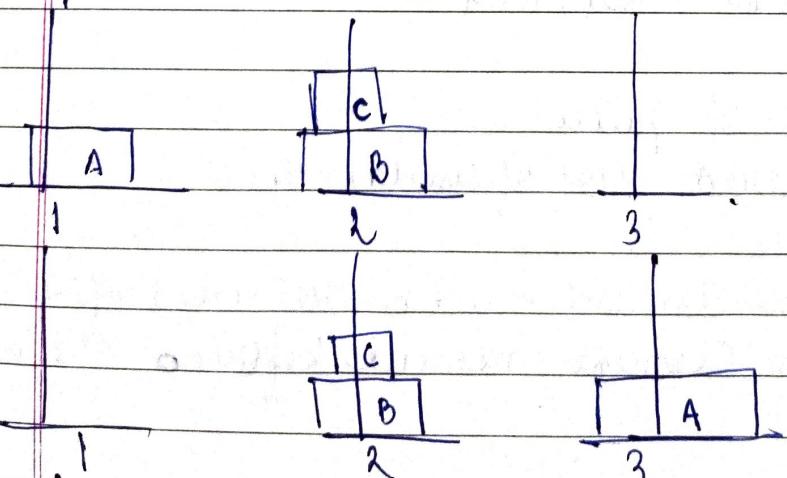
next

$3 \rightarrow 2$



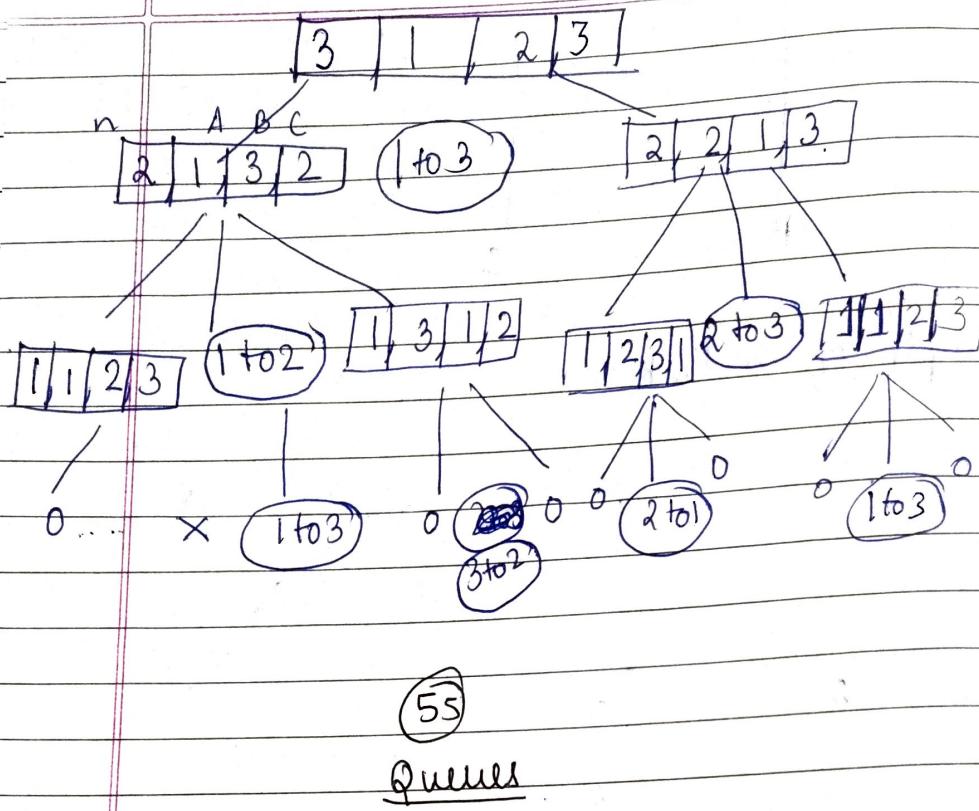
next

$1 \rightarrow 3$



no of moves = $2^n - 1$
no of function calls = $2^{n+1} - 1$

$(1, 3)$
 $(1, 2)$
 $(3, 2)$
 $(1, 3)$
 $(2, 1)$
 $(2, 3)$
 $(1, 3)$



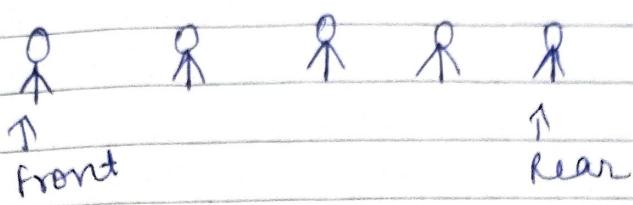
A Queue is a linear DS or linear list of elements in which deletion can take place at one end called 'FRONT' and insertion can take place at other end called 'REAR'.

Queue is also called as FIFO (first in first out) list.

Eg: Queue at movie ticket, ATM.

Basic features of Queue:

- 1) Queue is ordered list of similar type
- 2) FIFO structure
- 3) newly inserted element must be removed after removing the element inserted before the new element.

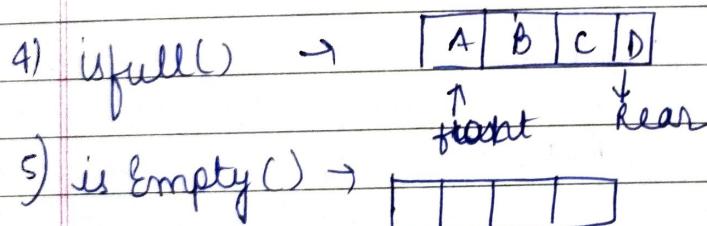


Applications of Queue

- 1) Sharing resource like printer, CPU scheduling
- 2) Call center (phone call)
- 3) Handling in real time system

Operations

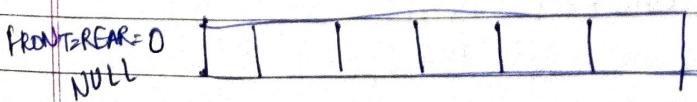
- 1) enqueue() → insertion (Rear)
- 2) dequeue() → deletion (front)
- 3) peek() → value of front (peak value)

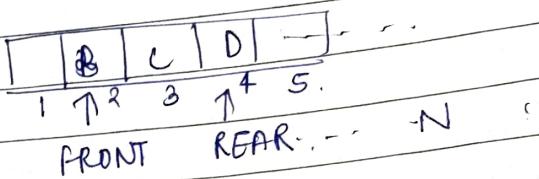
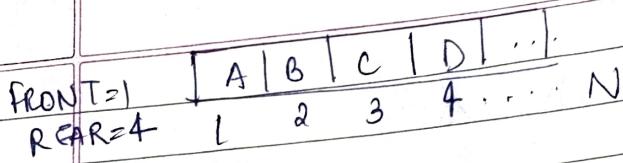


(56)

Array representation of Queue

Queue will maintain by a linear array with the help of 2 pointers "front" & "rear"



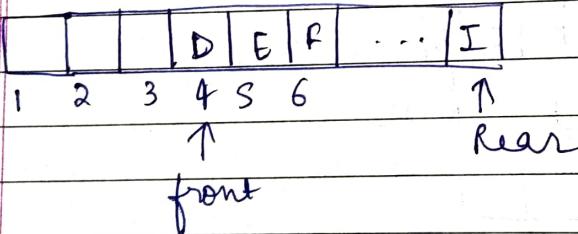


1) FRONT = NULL is empty \rightarrow queue is underflow

2) FRONT = FRONT + 1 (after deletion)

3) REAR = REAR + 1 (after insertion)

4) FRONT = 1 REAR = N (queue is full)
overflow.



queue is not full as front \neq 1

We do indexing of rear.

QJINSERT (Q, N, F, R, item)

1. if F = 1 and R = N or F = R + 1 then "overflow"

2. if F = NULL then F = 1, R = 1

3. else if R = N then set R = 1

4. else R = R + 1

5. Set Q[F] = item

6. Return

QDELETE(Q, N, F, R, Item)

1. if $F = \text{NULL}$ write "underflow"
2. set item = $Q[F]$
3. if $F = R$ then $F = \text{NULL}$ $R = \text{NULL}$
4. else if $F = N$ then set $F = 1$
5. else $F = F + 1$
6. Return

(57)

Linked list representation of LL.

A linked list is implemented using 2 pointers front & rear. Each node of linked list contain 2 part info & link

INSERT ALGO

1) Allocate space for new node
Pte.

2) Set $Pte \rightarrow \text{info} = \text{item}$

3) If $\text{front} = \text{NULL}$
Set $\text{FRONT} = \text{REAR} = Pte$

Set $\text{FRONT} \rightarrow \text{LINK} = \text{REAR} \rightarrow \text{LINK} = \text{NULL}$

4) Else

Set $\text{REAR} \rightarrow \text{LINK} = PTR$.

Set $\text{REAR} = PTR$.

Set $\text{REAR} \rightarrow \text{LINK} = \text{NULL}$

5) End

DELETE ALGO

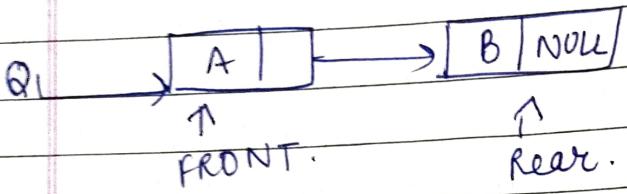
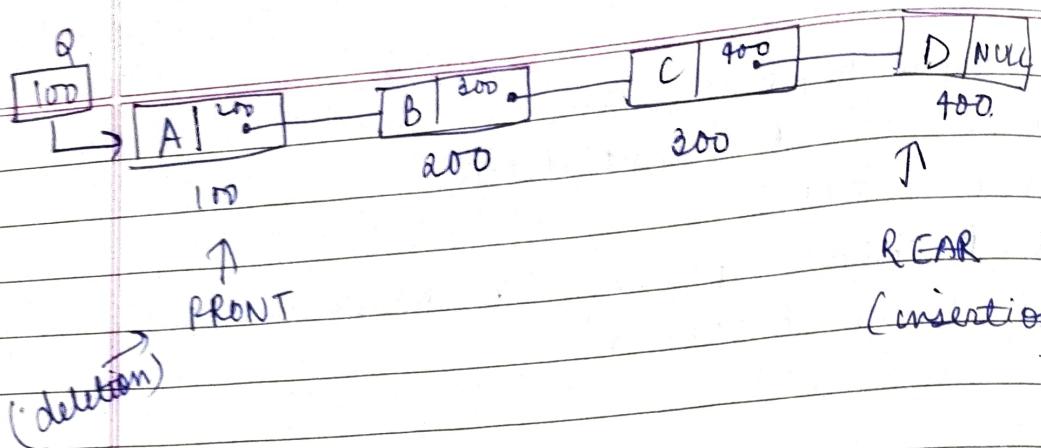
1) If $\text{FRONT} = \text{NULL}$ write underflow

2) Set $PTR = \text{FRONT}$

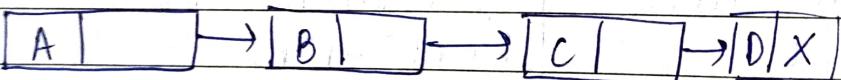
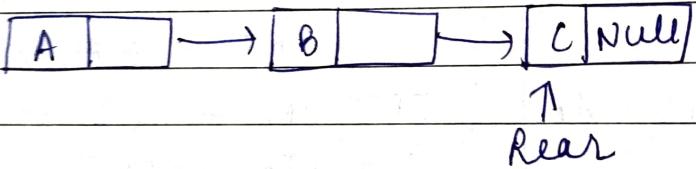
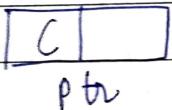
3) Set $\text{FRONT} = \text{FRONT} \rightarrow \text{LINK}$

4) FREEPTR.

5) End.



Insert c & D



5B

Types of Queues in D.S.

- 1) Simple Queue
- 2) Circular Queue
- 3) Priority Queue
- 4) Deque (Double ended Queue)

- 1) Simple Queue → We can insert at rear end & delete at front end. It follows LIFO rule.

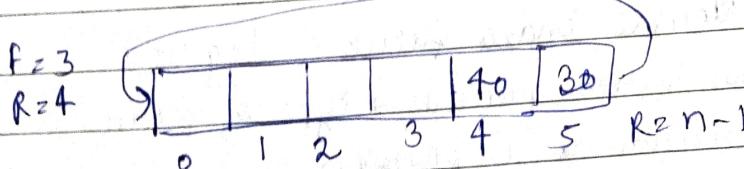
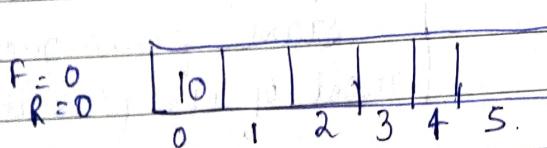
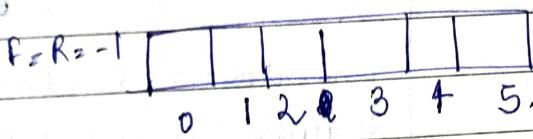
Circular Queue - In circular queue last position is connected to first position to make circular. The main advantage of circular queue is in utilization of memory.

INSERT

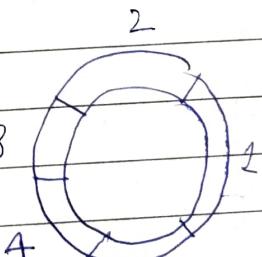
- 1) if $(R+1) \% N = F$, write overflow.
- 2) if $F = -1 \quad R = -1$ set $F = R = 0$.
- else set $R = (R+1) \% N$.
- 3) set $Q[R] = \text{item}$
- 4) exit

Delete

- 1) If $\text{front} = -1$ "write underflow"
- 2) Set $\text{item} = Q[F]$.
- 3) if $F = R$ set $F = R = -1$.
- 4) else, $F = (F+1) \% n$.
- 5) exit



$F=3$
 $R=0$



$F=1$
 $R=-1$

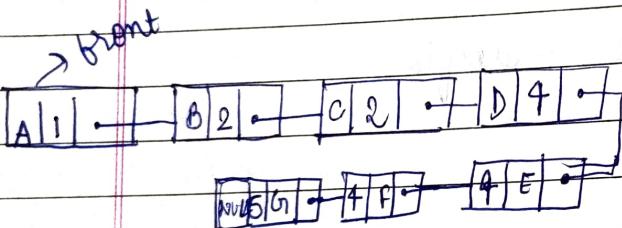
(59)

Priority Queue (Higher no; lower priority)

A priority queue is a collection of elements such that each element has been assigned a priority & serve acc'n to its priority.

1) Representation
linked list (one-way)

A	B	C	D	E	F	G
1	2	2	4	4	4	5



Deletion will take place from highest priority.

2) Array (multiple)

item	A	B	C	D	E	F	G
priority	1	2	2	4	4	4	5

$f = 0$	$f = 0$	0	1	2	3	4	5	$P = 0$
Q_1	A							$R = 0$
Q_2	B	C						$P = 0$
Q_4	D	E	F					$R = 0$
Q_5	G							$P = 0$
	0	1	2	3	4	5		$R = 0$

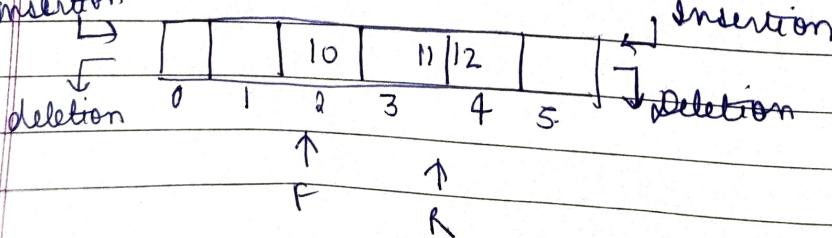
Insertion & deletion using simple queue. Deletion starts from queue having most of the priority.

(60)

Double Ended Queue (dequeue)

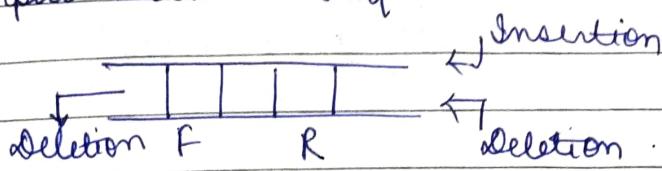
Dequeue or double ended queue is a type of queue in which insertion & deletion can be performed from either FRONT or REAR. It doesn't follow FIFO rule.

Insertion



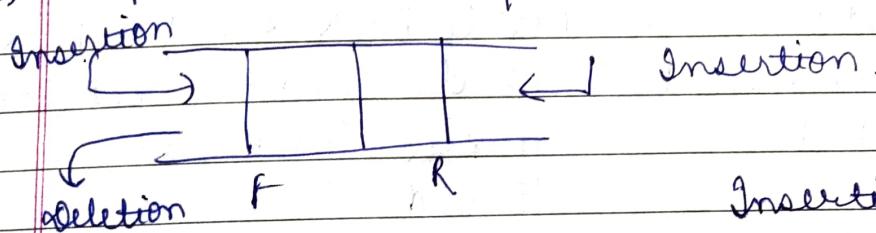
- 1) Insert at FRONT
- 2) Insert at REAR
- 3) Delete from FRONT
- 4) Delete from REAR

a) Input restricted Queue



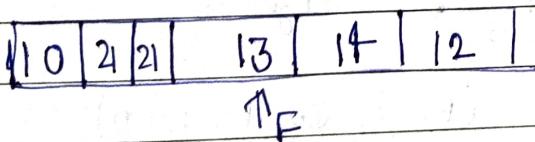
Insertion → Rear side
Deletion → front as well as rear.

b) Output - restricted queue.



Insertion → Front as well as rear
Deletion → ~~Front~~ front side

Insertion at front takes a circular manner.



Insert at front

10

12

14

13

Insert at Rear

21

22.

Insert FRONT

1. If $F = 0$ and $R = N - 1$ or $F = R + 1$
write "overflow"
2. If $F = -1$ set $F = R = 0$.
3. else if $F = 0$, set $F = N - 1$
4. else set $F = F - 1$
5. $Q[F] = \text{item}$

Insert REAR

- 1) If $F = 0$ & $R = N - 1$ or $F = R + 1$
write "overflow"
- 2) If $F = -1$ set $F = R = 0$.
- 3) Elseif $R = N - 1$ set $R = 0$
- 4) Else $R = R + 1$
- 5) $Q[R] = \text{item}$

Delete FRONT

1. If $F = -1$ write "underflow"
 2. If $F = R$ set $F = -1$, $R = -1$
 3. Elseif $F = N - 1$ set $F = 0$
 4. Else $F = F + 1$
 5. End
1. If $F = -1$ write "underflow"
 2. If $F = R$ set $F = -1$, $R = -1$
 3. Elseif $R = 0$ set $R = N - 1$
 4. Else $R = R - 1$
 5. End

Delete REAR

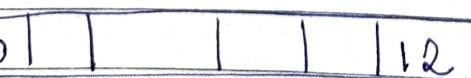
$F = -1$  size = 6.

$R = -1$

insert 10 at front ($F = -1$ so $F = R = 0$)

$F = R = 0$  size = 6

insert 12 at front ($F = 0$ so $F = n - 1 = 5$)

$F = 5$  size = 6

$R = 0$

insert 14 at second

$F=4$	10				14	12	$(F=F+1)$
$R=2$	0	1	2	3	4	5	

insert 21 at rear.

$F=4$	10	21	.		14	12	$(R=R+1)$
$R=2$	0	1	2	3	4	5	

delete from front 14

$F=5$	10	21	.		14	12	$(F=F+1)$
$R=2$	0	1	2	3	4	5	

delete from front 12

$F=0$	10	21	.		12		$F=N-1$
$R=2$	0	1	2	3	4	5	$\leftarrow F$

$\text{let } F=0$

delete from rear 21

$F=0$	10						$R=R-1$
$R=1$	0	1	2	3	4	5	

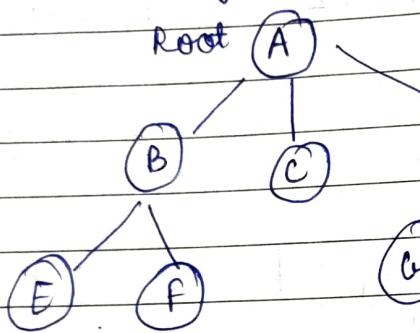
(61)

Trees.

Tree is a non linear ds. This is used to represent hierarchical relationship betn. elements.

Properties

- 1) There are only one root no parent
- 2) Except root each node have exactly one parent
- 3) A node may have zero or more children
- 4) There are unique path from root to each node.
- 5) There is no cycle created in tree.

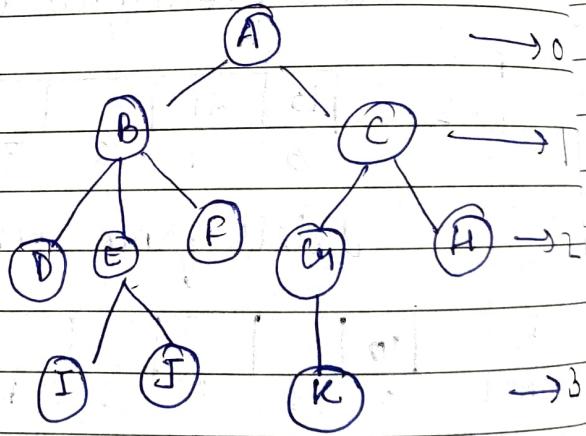


Nodes: A, B, C, D, E, G, H

Edges: (A-B), (A-C), (B-E), (B-F), (D-G), (D-H)

Terminologies

1. Root: Top of tree from which it grows. Ex - A



2. Parent node: Having atleast one children.

Ex - A, B, C, E

3. Child node : Every node having a parent

Ex - B, C, D, E, F, G, H, I, J, K

4. Leaves : no children.

Ex - D, I, J, K, H

5. Subtree

6. degree

A

B

D

E

F

7. degree
no

8. level

9. He

H

5. Subtree : Subtree (division of tree)

6. degree of node : how many child nodes
 d

$A \rightarrow 2$	$C \rightarrow 2$
$B \rightarrow 3$	$D \rightarrow 1$
$D \rightarrow 0$	$H \rightarrow 0$
$E \rightarrow 2$	$K \rightarrow 0$
$F \rightarrow 0$	$I \rightarrow 0$

7. degree of tree : highest degree of degree of node.

Ans $d = 3$.

8. level of tree: Root --- level 0

--- level 1

level 2

9. Height and depth of tree

Height of A = leaf node se pahuchne ka longest path. (3)

Height of B = 2.

$D = 0$

$C = 2$

$K = 1$

Depth of B = root se kitna edge hai B per pahuchne ka.

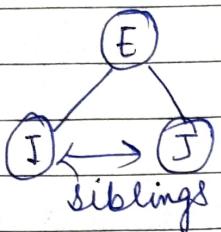
Dept of K = 3

Dept of A = 0

10. Internal node : having children

11. External node : no children (leaf node)

12. Sibling: If a node has more than 1 children



Ex: $I \rightarrow J$, $C \rightarrow H$

13. Ancestor and proper ancestor : If we want to find ancestors for D i.e. root se D tak pahuchne me kitne nodes aa rakte hai.

ancestor for D (ancestor) : A, B, D (including D)

proper ancestor for D (proper ancestor) : A, B (excluding D)

14. Descendent & proper descendent : From the given node to the leaf node.

for B (descendent) : D, E, F, I, J, B (including B)

for D (proper descendent) : D, E, F, I, J (excluding B)

Types of tree

- 1) Binary tree.
- 2) Binary search tree.
- 3) AVL tree
- 4) B-tree.

Date: / /
Page No. / /

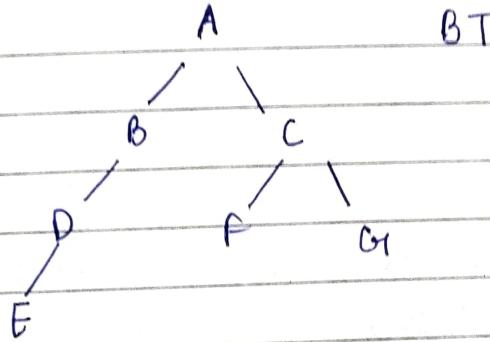
(62)

Binary Tree.

A binary tree is a spcl kind of tree in which every node maxm. of 2 children. Any node in binary tree has either 0, 1 or 2 children.

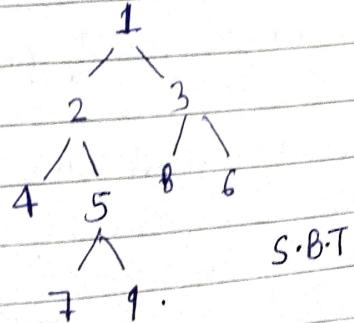
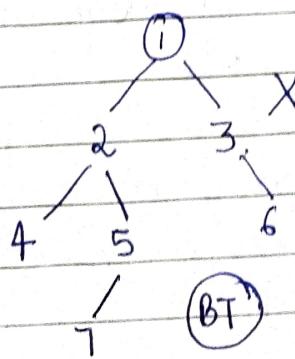
Types of Binary tree

- 1. Strictly BT (full/proper)
- 2. Complete BT
- 3. Perfect BT
- 4. Balanced BT
- 5. Extended BT



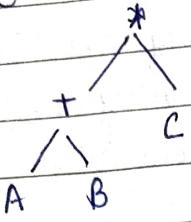
$$\begin{array}{lll} A=2 & D=1 & G=0 \\ B=1 & E=0 & \\ C=2 & F=0. & \end{array}$$

- 1. Strictly BT - In strictly BT every node should have exactly 2 children all none.
A BT in which every node has either 2 or zero children.

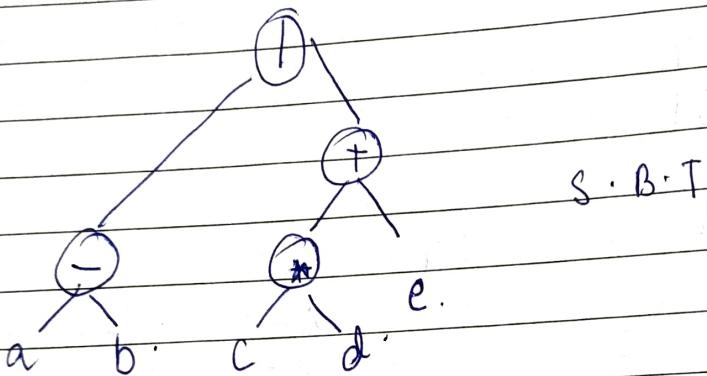


In case of arithmetic calcⁿ.

$(A+B)*C$



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



operators \rightarrow internal nodes

variables \rightarrow external nodes

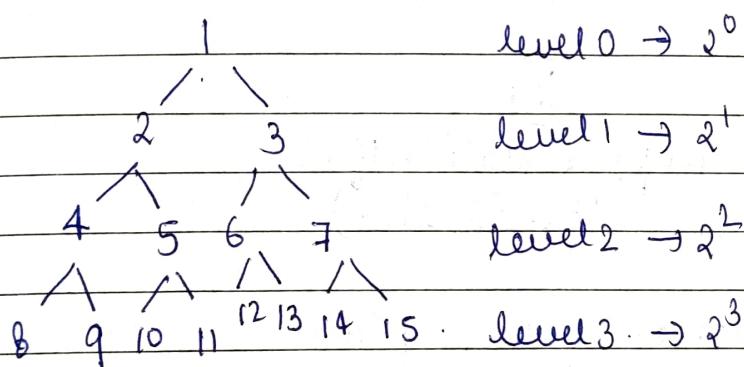
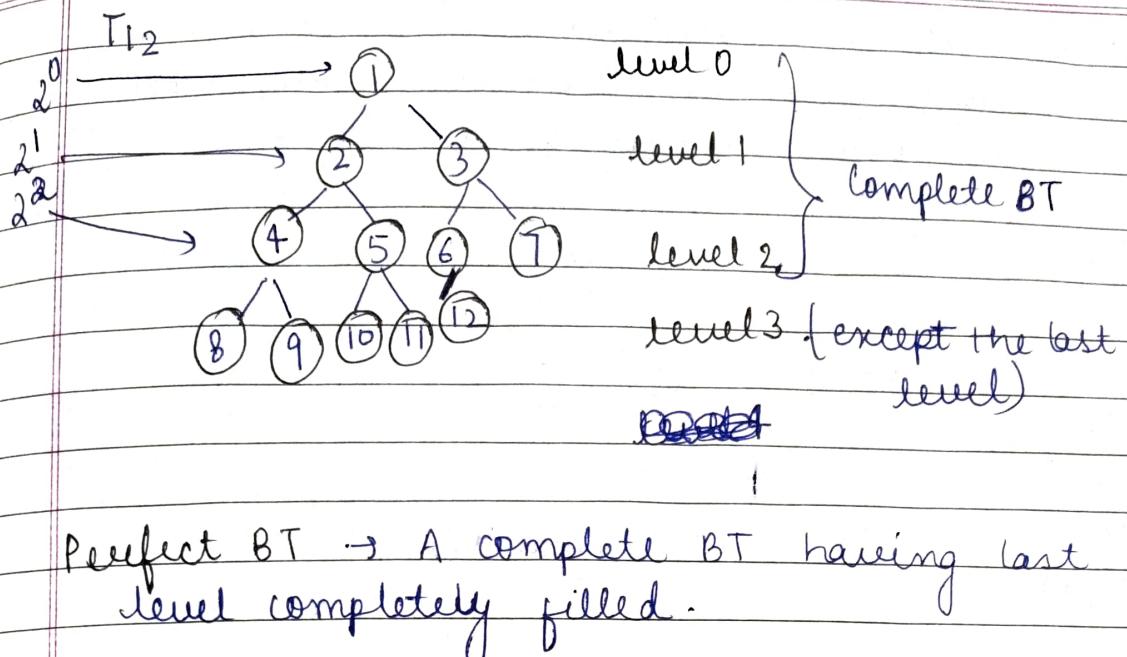
(63)

Complete BT

2. Complete BT : The BT is said to be complete if all its level except possibly the last have max^m no of possible node.

Notes: 1. Each level have at max 2^r node. r is level.

2. The path of complete BT T_n with node n is given $P_n = \lceil \log_2 n + 1 \rceil$



(64)

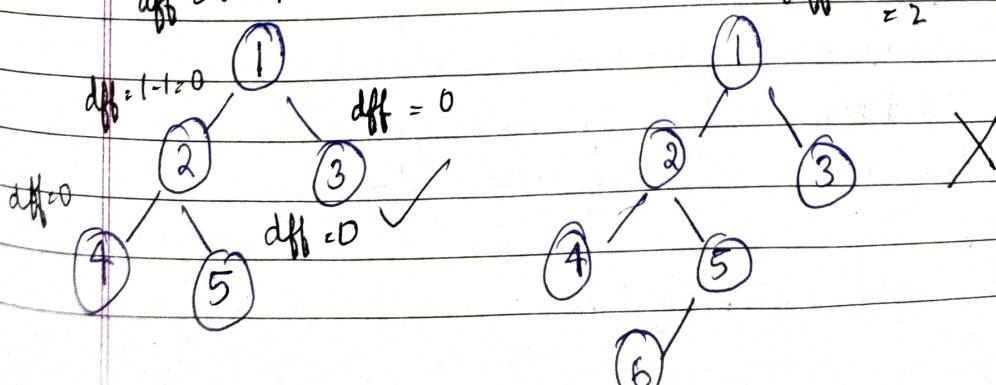
Balanced & Extended BT

Balanced BT \rightarrow It is defined as BT in which the height of left & right subtree of any node differ by not more than 1.

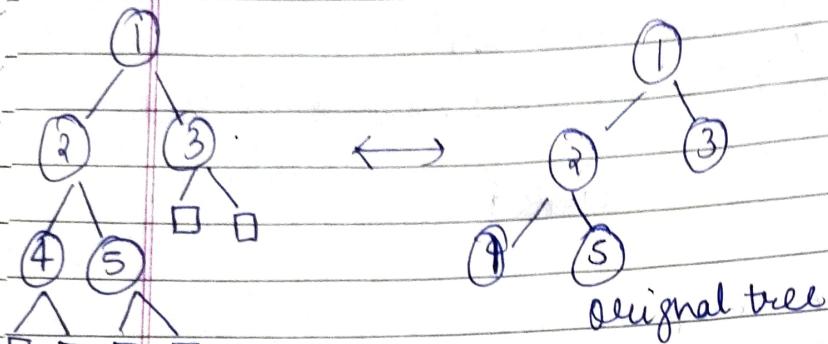
$$df = |\text{height of left child} - \text{height of right child}|$$

$$df = 2-1=1$$

$$df = 3-1 = 2$$



(2tree) Extended BT - Extended BT is a type of BT in which all null subtrees at original tree are replaced with special nodes called external node.



(65)

Representation of BT (linked list)

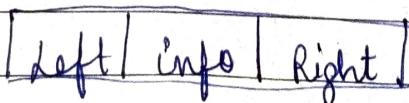
Representation of Binary tree

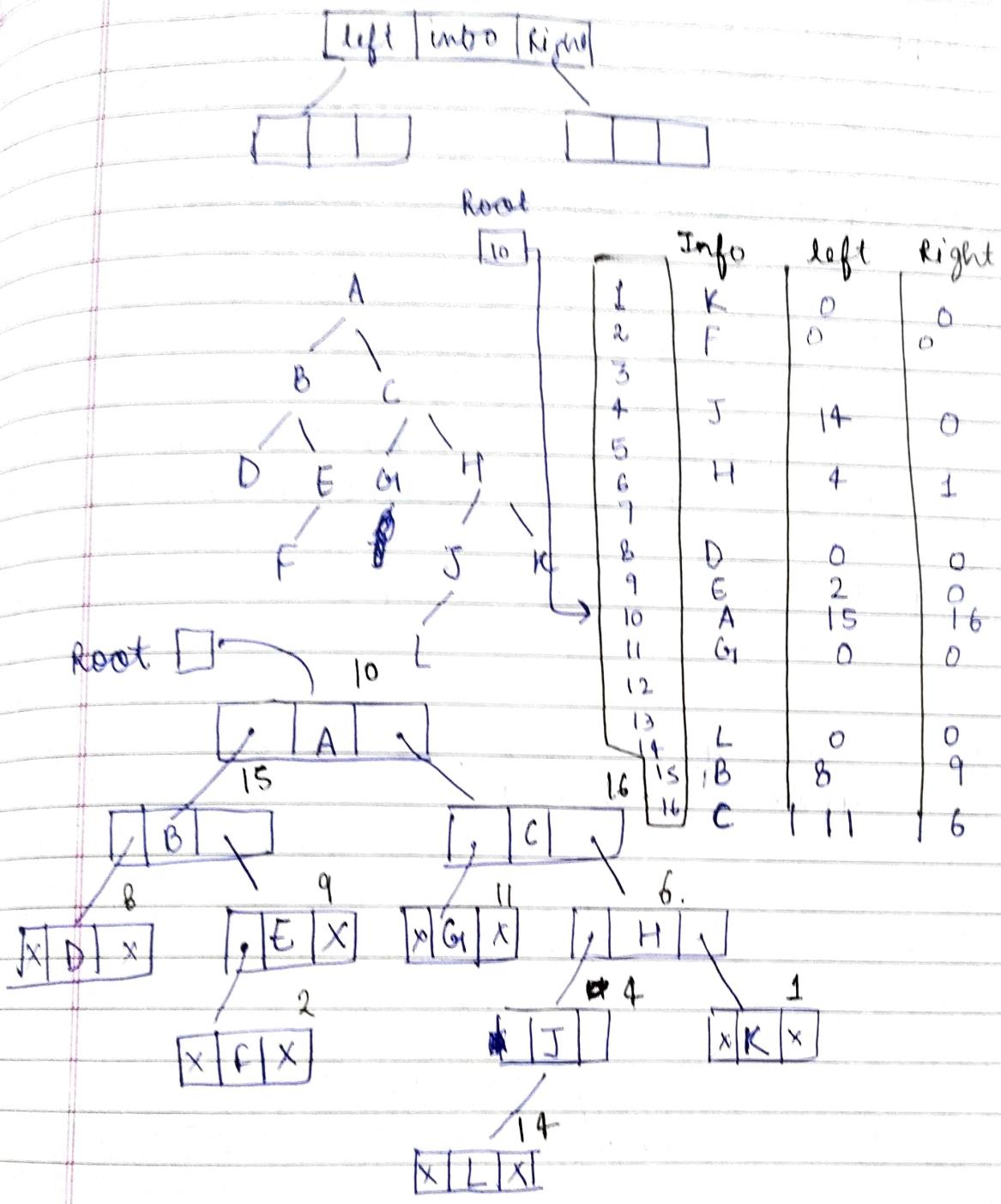
- 1) Link representation (linked list)
- 2) Sequential representation (Array)

Using LL representation

BT will maintain in memory by LL representation using 3 parallel array:

- 1) INFO → contain data at node.
- 2) LEFT → contain location of left child.
- 3) RIGHT → contain locⁿ of right child.
- 4) ROOT → contain locⁿ of root of tree.





(66)

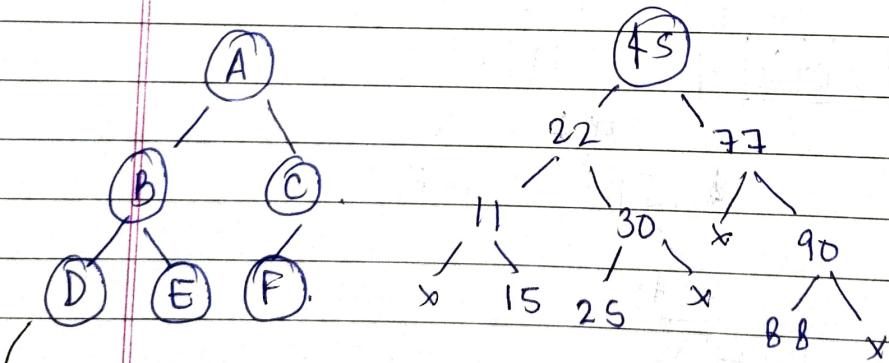
Sequential Representation (Array)

Suppose T is a BT that is complete BT then there is an efficient way to maintain in memory called sequential representation.

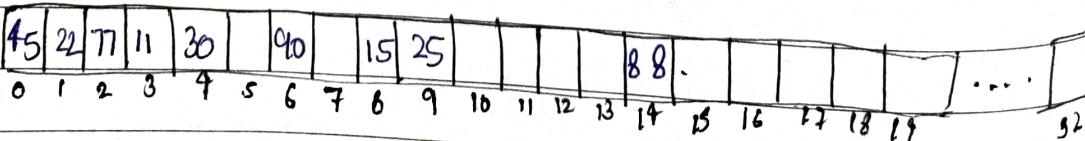
This representation uses simple linear array as:

1. The root of tree stored at $T[1]$
2. If node N occupies $T[k:j]$ then its left child is stored at $2k$ & right child at $2k+1$.
3. If $T[i] = \text{NULL}$ then tree is empty.

Note: A tree with depth d will require an array with approx 2^{d+1} element



Array doesn't have dynamic memory allocation i.e. why wastage of memory takes place

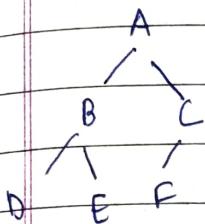
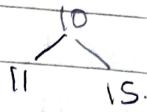


memory wastage

$T[k]$	
1	10
2	11
3	15
4	
5	

$$2 \times k = 2 \times 1 = 2$$

$$2 \times 1 + 1 = 2 \times 1 + 1 = 3$$



~~2~~

$$2^{d+1} = 2^{3+1} = 2^4 = 16$$

A	B	C	D	E	F										
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1
2
3
4
5

$$B = 2x^2 + 4 \text{ (LC)}$$

Ex 3: 6. (LC)

$$5 \quad 2 \times 2 + 1 = 5 \text{ (RC)}$$

67

Teameering BT

Jeee

- 1) Pre-order (Node - left - Right)
 - 2) In-order (left - node - Right)
 - 3) Post-order (left - Right - node)

Pre-order

1) Process the Root R.

2) Traverse the left subtree of R
in pre-order

3) Traverse in the right subtree
of R in pre-order

Post-order.

1) Traverse the left subtree

2) Traverse the right subtree

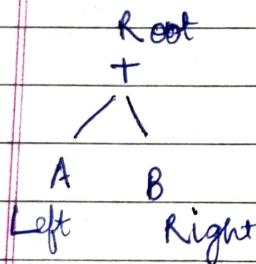
3) Process the Root R.

In-order

1) Traverse the left subtree of R

2) Process the Root

3) Traverse the right subtree of
R in in-order.



+ A B.

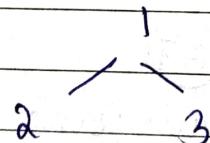
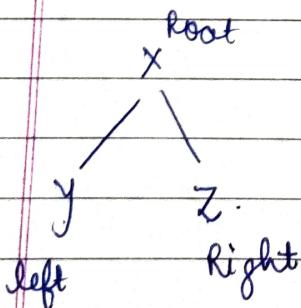
A + B.

ABT

prefix / preorder X Y Z.

infix / in-order Y X Z

postfix / post-order Y Z X

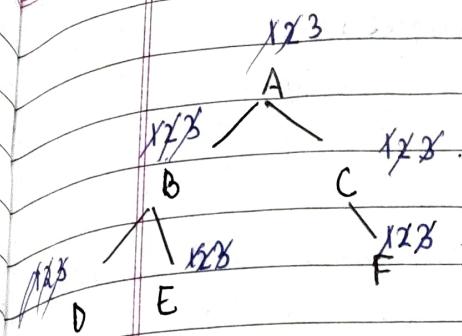


1 2 3

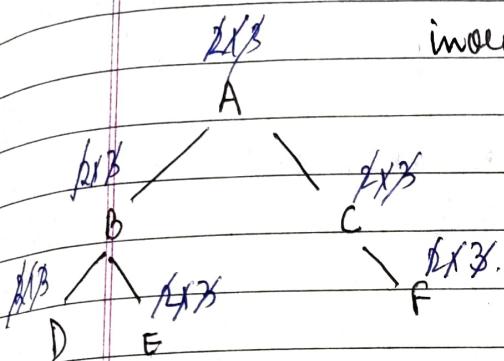
2 1 3

2 3 1

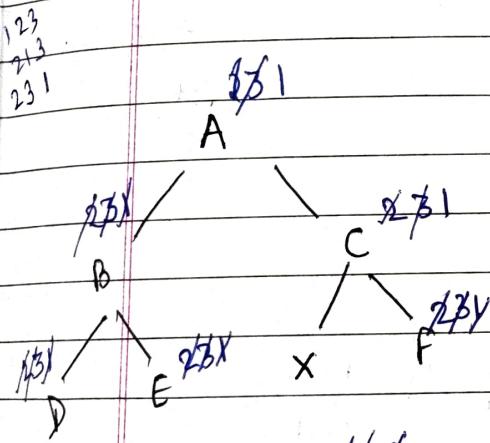
Now let's find preorder, post-order and
in-order.



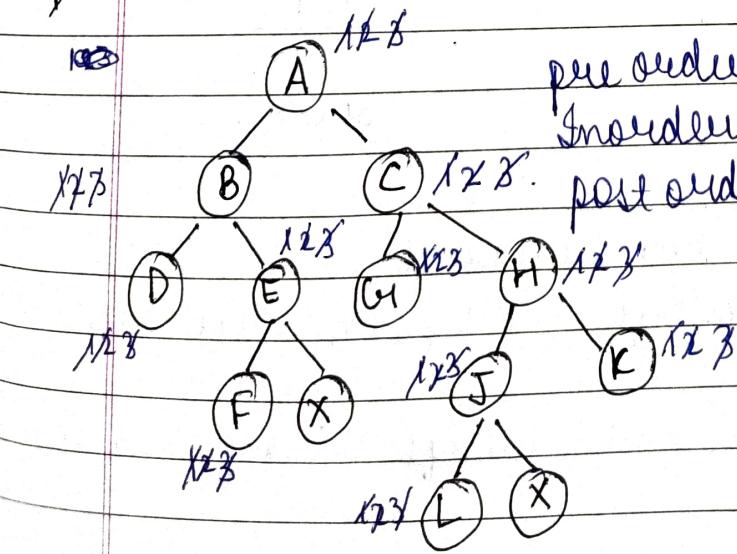
inorder: ABDEC F



preorder: DBEAC F



postorder: DEBFCA



preorder (123): ABDEF CGHJKL

inorder (213): DBFEAGCHJKL

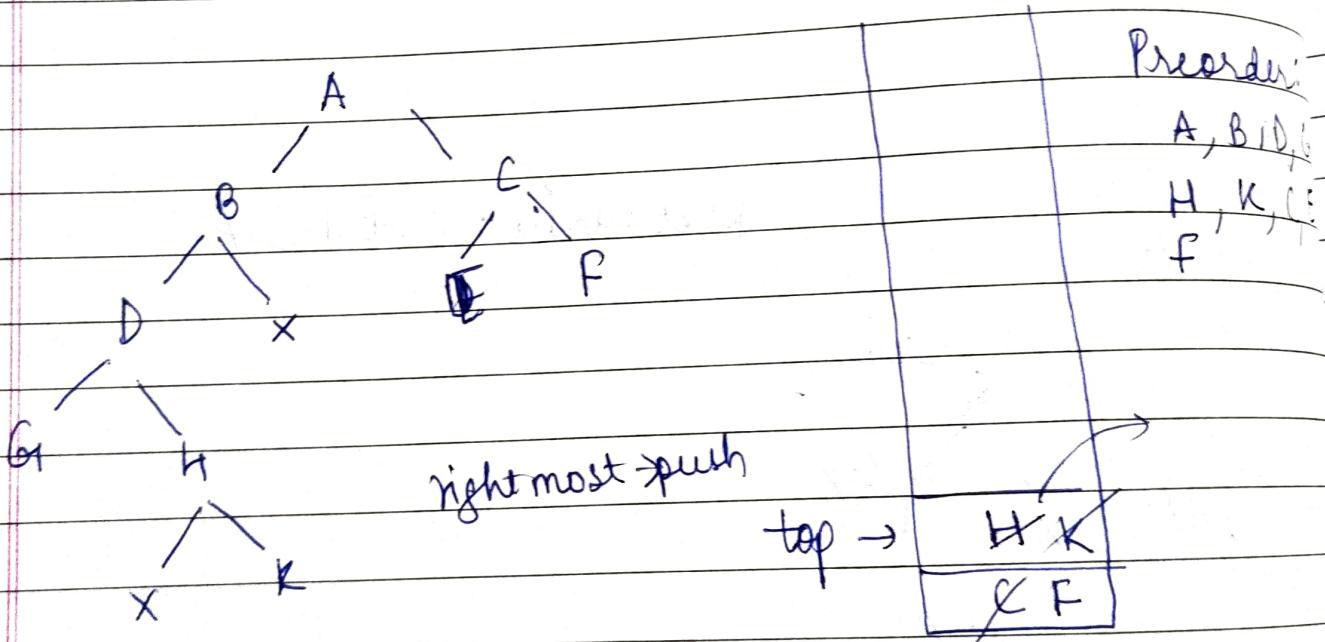
postorder (231): DFEBGILJKHCA

Preorder Tree Traversal

Traversing of BT using stack

Preorder.

- 1) Proceed down to left most path, processing each node N on path and push each right child onto the stack. The traversing ends after node N with no left child processed.
- 2) Pop top element on stack, then return to step a if stack is empty exit.



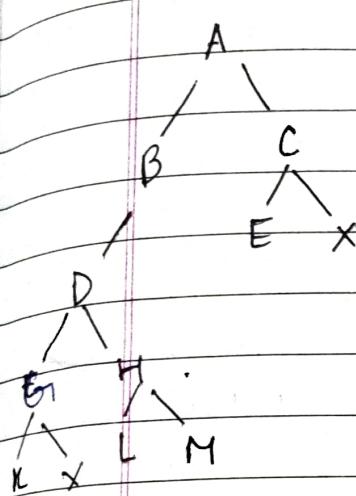
(69)

Inorder tree traversal.

Proceed down to left most path Push each node N on the stack

Stop when node n with no left child pushed on stack.

- b) Pop & process the node on stack.
- 1) if null is popped exit
 - 2) if a node N with right child is processed & return to step (a).



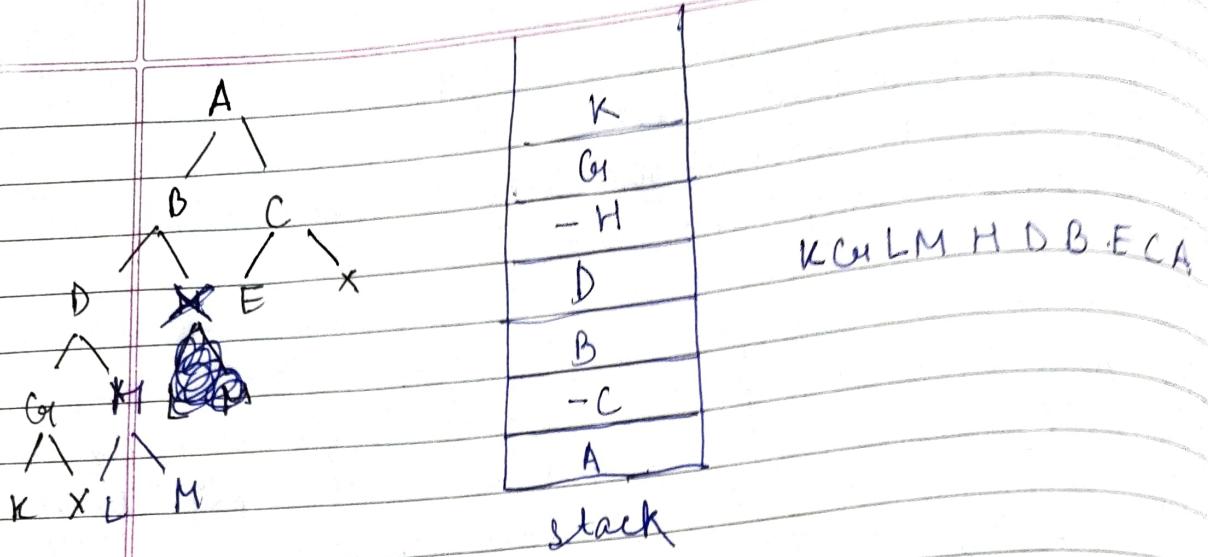
		leftmost \rightarrow push then pop till right chd.
		Inorder:
	K X	K, O, D, L, H, M, B, A, E
	G ¹ X L ² X	
	M ³ X D ¹ X H ² X	
	B ¹ X E ² X	
	A X C ² X	

Phle leftmost ko insert karte jao. Phir tab tak pop kro jab tak us node ka right child na ho. Phir right child node ke baki left most nodes ko ekhte jao and again the same.

(69)

Post order tree traversal

- a. Proceed down to left most path pushing each node on stack if N has a right child push "right child".
- b. pop and process positive node on stack.
- c) if negative node is popped return to step a.
- d) if null is popped, exit.



left child insert karte jaos agar risi parent ka
right child hai to use stack me - right child ke
with push kro.

Jaise -ve node aae phirse left most node ko
traverse kro. jo right node hai jaise
ki H phir - M and L.

(II)

Construction of Binary tree from
given traversal.

- Binary tree from pre & inorder
- " " " post & inorder
- " " " pre and post

Steps

- Identify root from preorder.
- Identify element left and right subtree from inorder.
- Repeat step 1 & 2.

pre order : 1 2 4 3 5 7 8 6

inorder : 4 2 1 7 5 8 3 6

left subtree right subtree



123

213

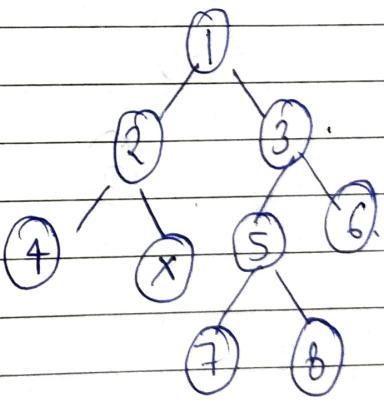
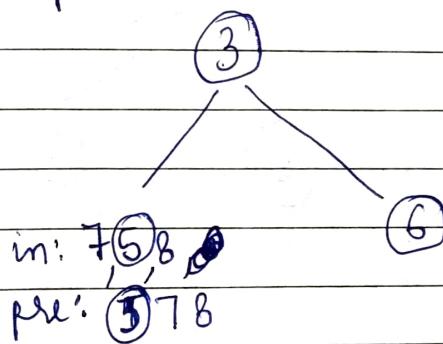
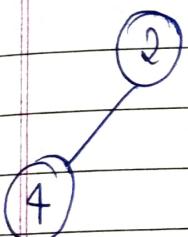
~~231~~

231

in: 4, 2^{root}
pre: 2, 4

in: 7 5 8 3 6.
pre: 3 5 7 8 6.

Root



(12)

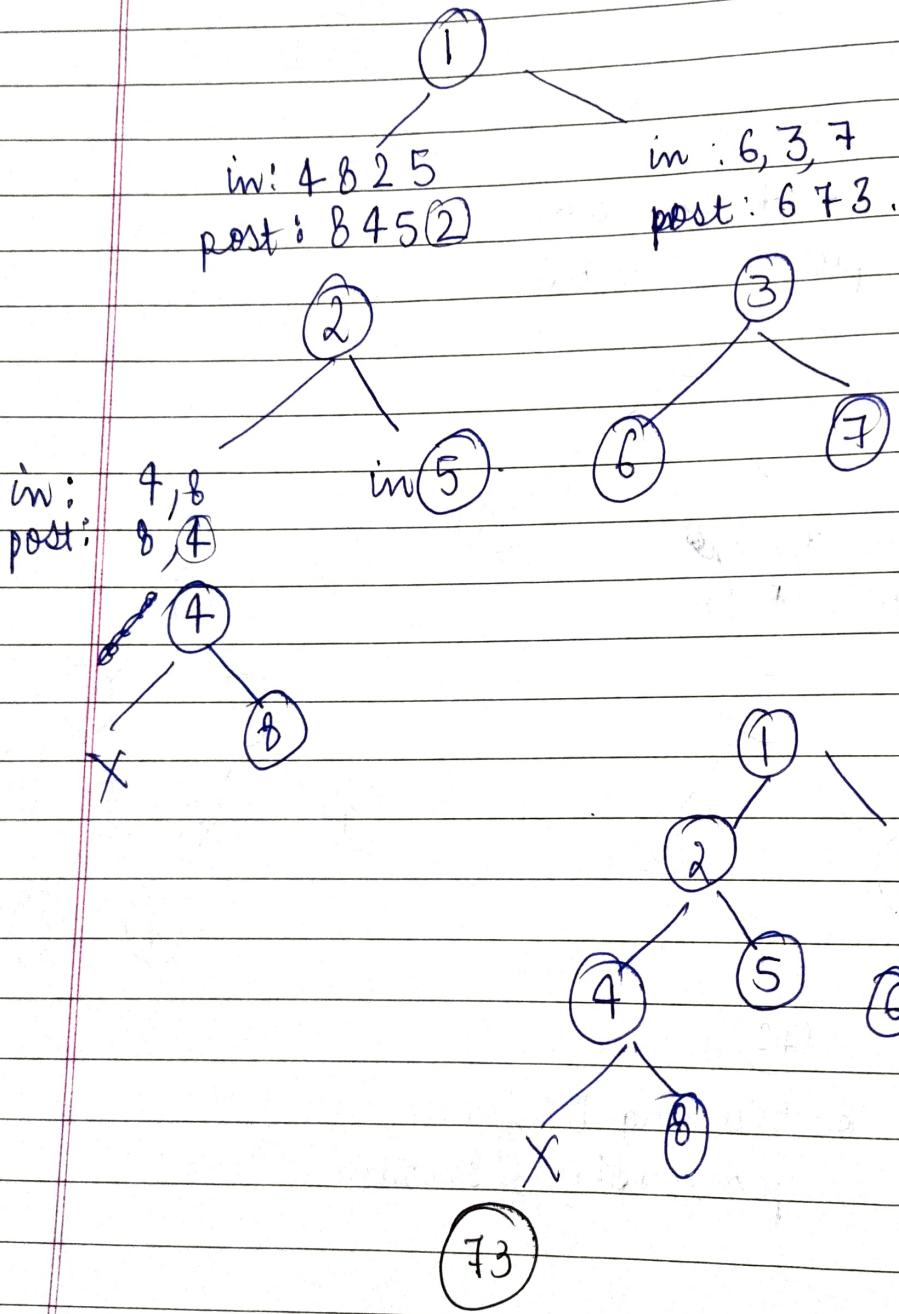
Constructing BT from postorder & inorder.

Steps :

- 1) Identify root from post order
- 2) Identify elements of left & right subtree from inorder
- 3) Repeat step 1 & 2.

Postorder: 8, 4, 5, 2, 6, 7, 3, 1
Inorder: 4, 8, 2, 5, 1, 6, 3, 7

Post : 231
In : ~~231~~
213



Constructing from pre & post order.

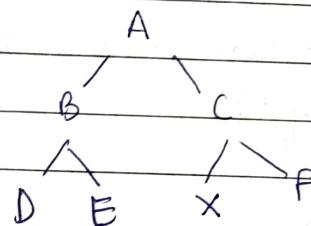
Steps

- 1) Identify root from pre order
- 2) Identify left child from pre order
- 3) Identify left subtree & right subtree from post order
- 4) Recursively repeat steps for each subtree

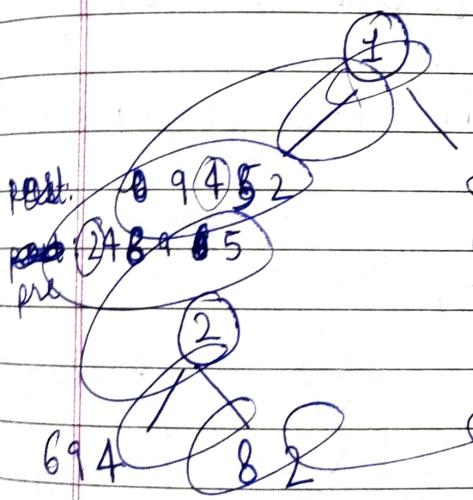
always full BT
Pre order: 1, 2, 4, 8, 9, 5, 3, 6, 7

Post order: 8, 9, 4, 5, 2, 6, 7, 3 ①

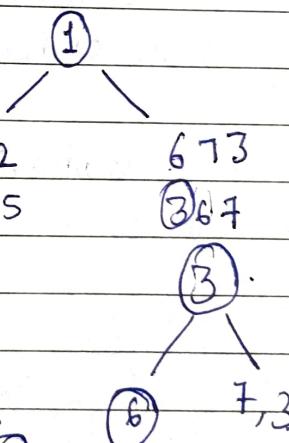
Pre 123
Post 231



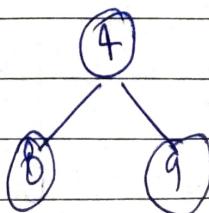
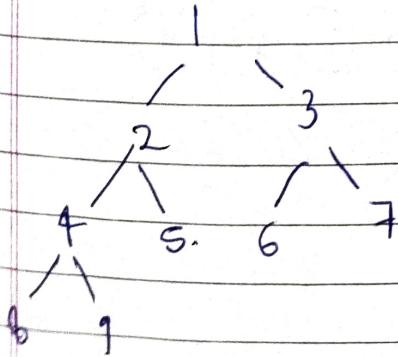
Preorder: AB¹DECF
Postorder: DEBFCA
left Right



post: 89452
pre: 24895



post: 894
pre: 489



74

Binary Search Tree.

binary search tree every node is organized in specific order. This is also called ordered binary tree.

Suppose T is binary tree then T is called BST if each node of T has following properties.

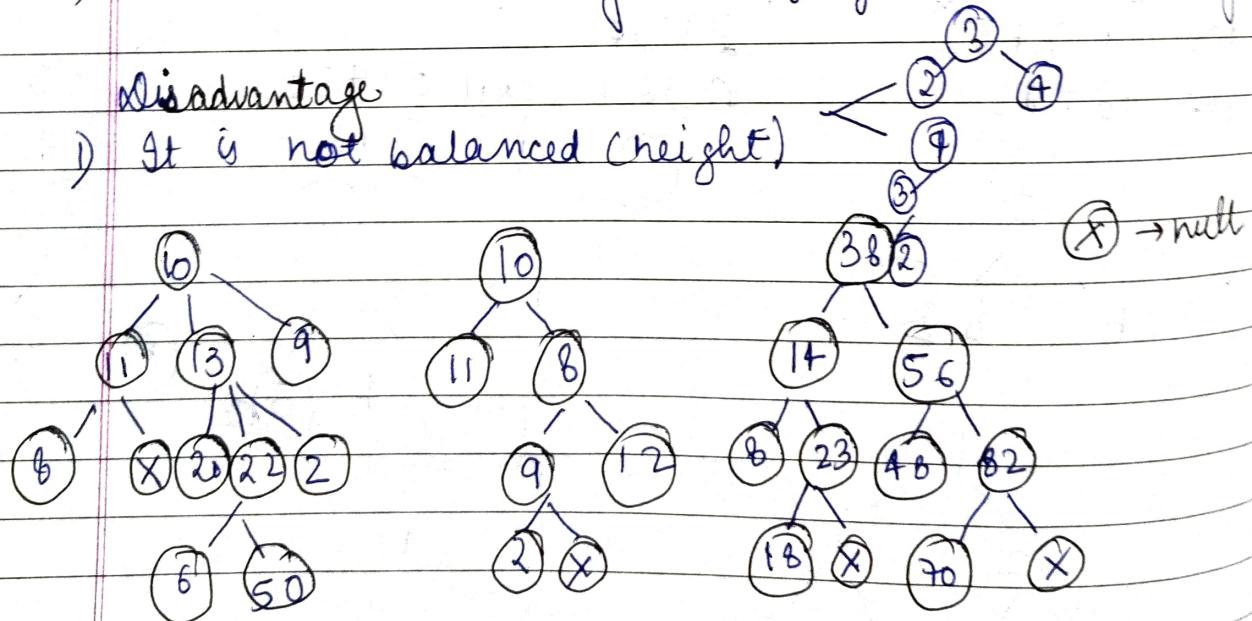
1. The value at N is greater than every value in left subtree.
2. And N is less than every value in Right subtree of N .

Advantage

- 1) Avg time of searching, inserting, deleting of element in BST is $O(\log n)$
- 2) In order traversing always gives sorted array

Disadvantage

- 1) It is not balanced (height)

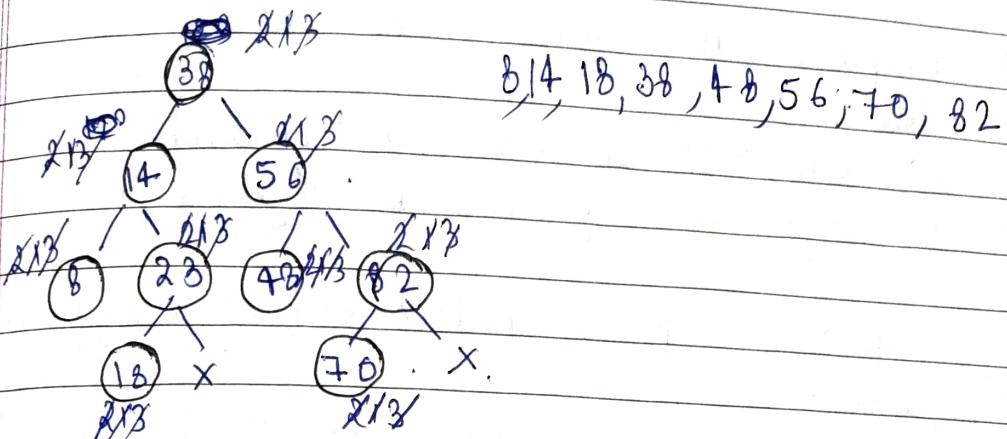


General Tree

Binary tree

BST

Inorder

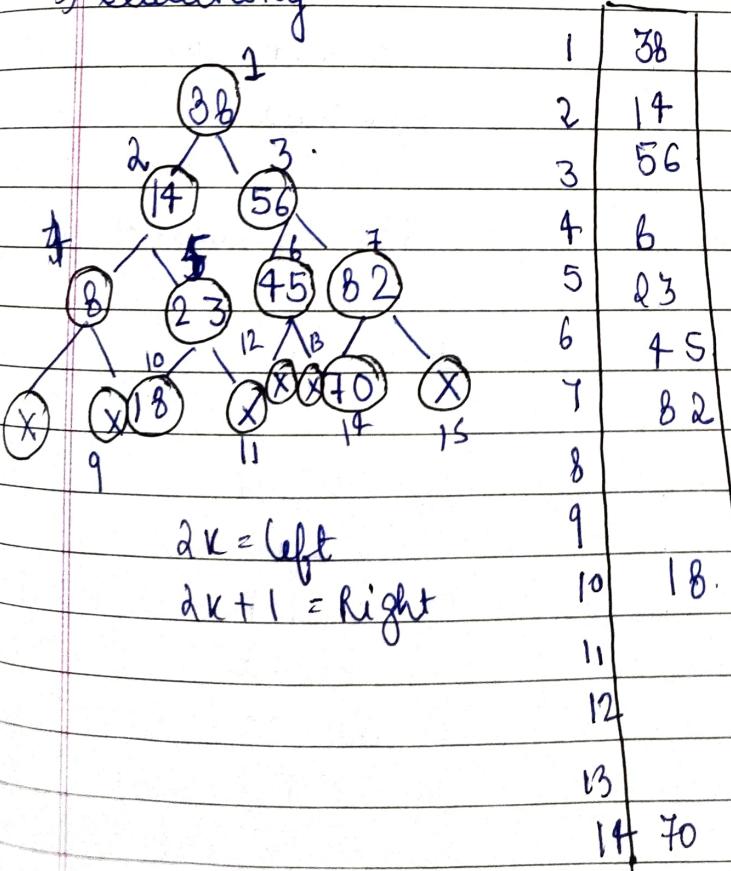


7.5

Binary Search Tree
Insertion.

Operations performed on BST

- 1) Insertion
- 2) Deletion
- 3) Searching



Searching

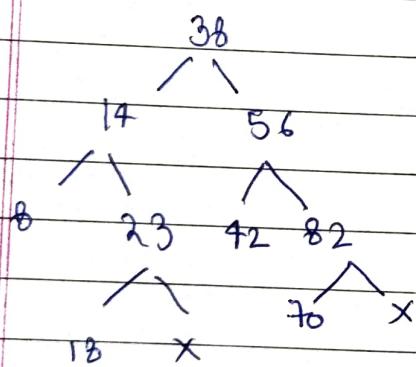
Step a: Compare item with root node n .

if item $< n$, proceed to left child.

if item $> n$ proceed to right child

Step b: Repeat step (a) until one of the following occur:

- i) We meet a node N such that item $= n$
- ii) we meet an empty subtree which indicate unsuccessful and insert item at empty location.



item = 23

a) $23 < 38$

left side

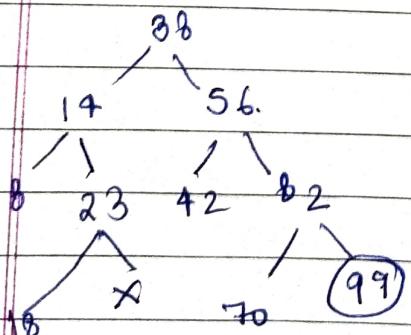
b) $23 < 14$ or $23 > 14$

right child

Case 1:

To insert 99.

c) $23 = 23$ [Search success]



item = 20

a) $20 < 38$

left

b) $20 < 14$ or $20 > 14$

right

c) $20 < 23$ left

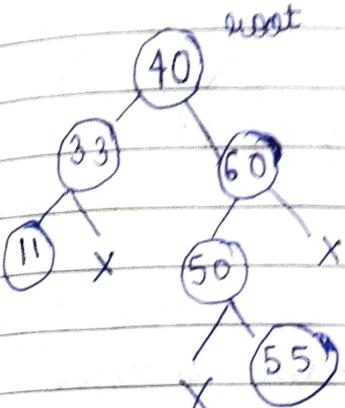
d) $20 > 18$ right

e) $20 (=)$ null element
not found

Case 2:

Q. Insert into empty BST in order.

40, 60, 50, 33, 55, 11

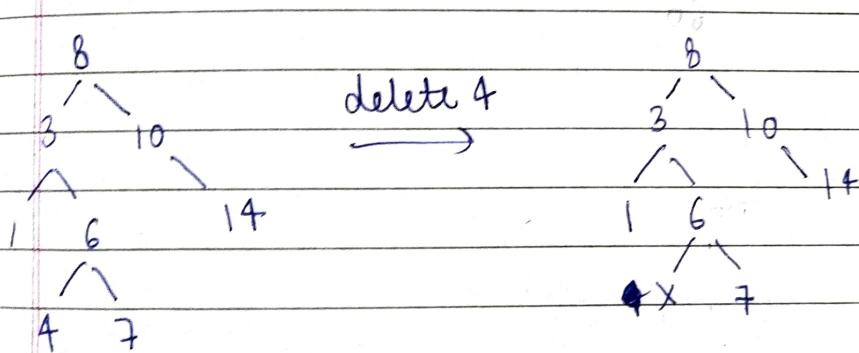


(76)

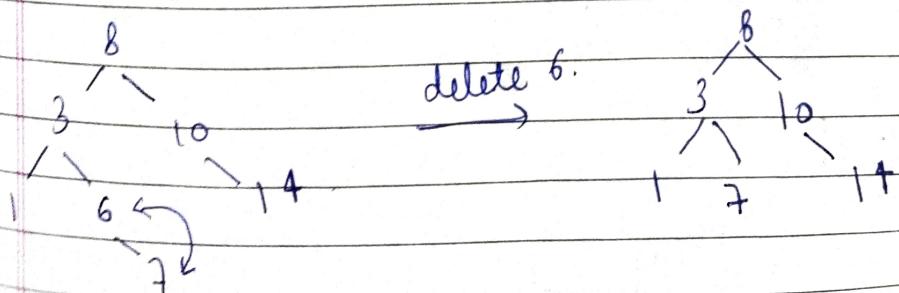
Deletion in BST

Case 1: Node is a leaf node.

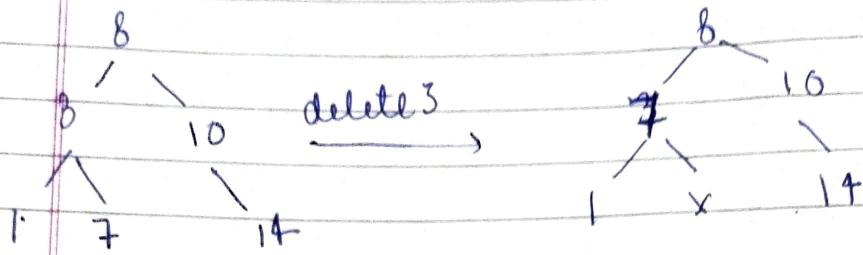
Node if a leaf node is simply removed



Case 2: Node has a single child.



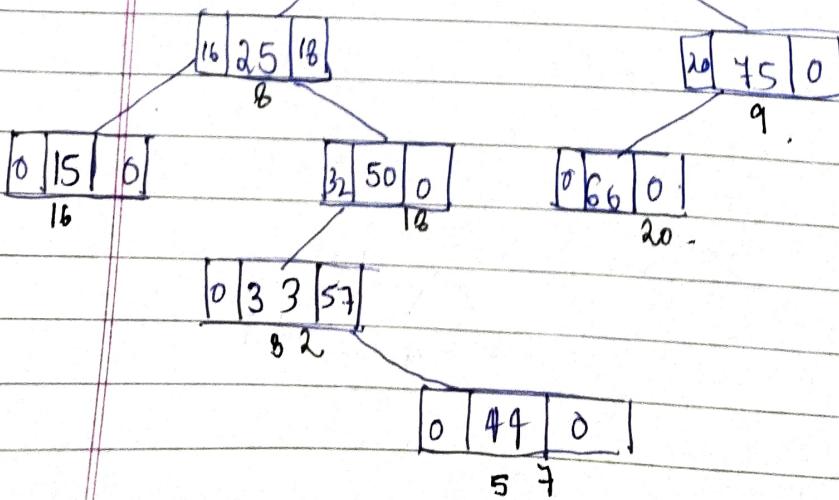
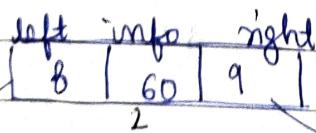
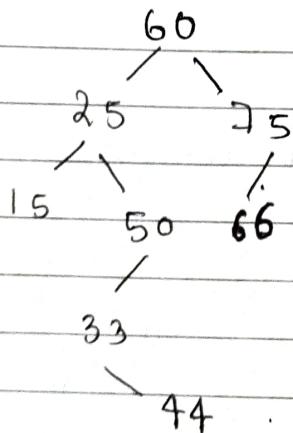
Case 3: Node having 2 children.



We will find the inorder successor & replace the element by its inorder successor.

Inorder : 1 3 7 8 10 14

↑
inorder
successor.



delete 44

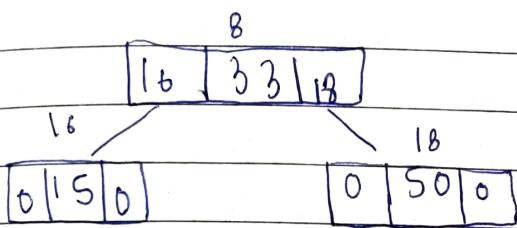
0	33	0
---	----	---

delete 77

6	60	20
---	----	----

delete 25.

inorder: 15 25 33 50 60 66



(77)

AVL trees

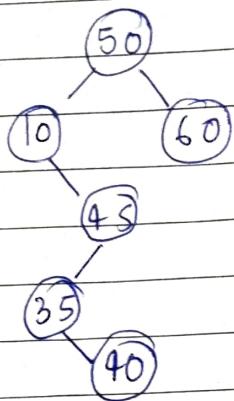
AVL tree can be defined as height balanced binary search tree in which each node is associated with balance factor (BF) bet^n. (-1 to 1) or either -1, 0, or 1. AVL tree introduced in 1962 by Adelson - Velskii - Landis

Balance factor = height of left subtree - height of right subtree

Problem with BST

1)

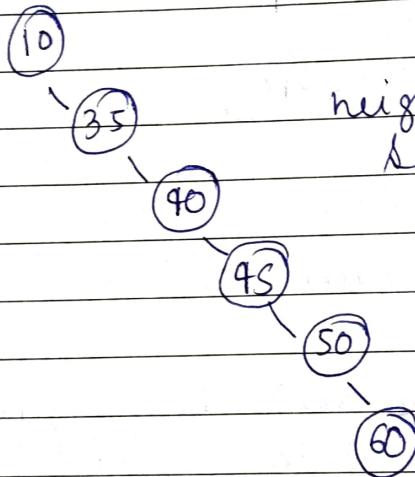
50, 10, 45, 60, 35, 40.



height = 4

Search $\Rightarrow O(\log N)$

10, 35, 40, 45, 50, 60



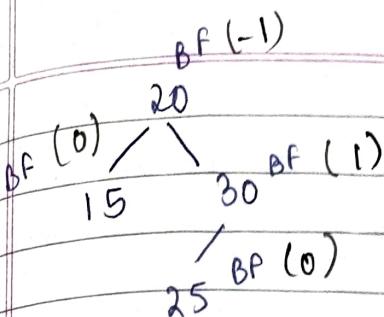
height = 5

Search $= O(n) \rightarrow \text{max}$

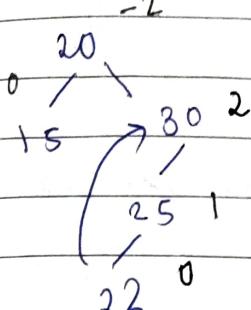
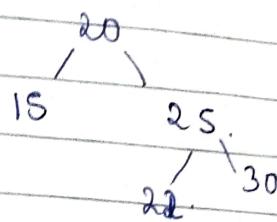
Tree is not height balanced. AVL tree can help in this.

Rotations

- 1) Left-left rotation (LL)
- 2) Right-right rotation (RR)
- 3) Left-right rotation (LR)
- 4) Right-left rotation (RL)

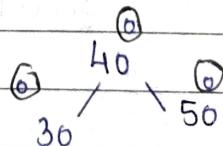
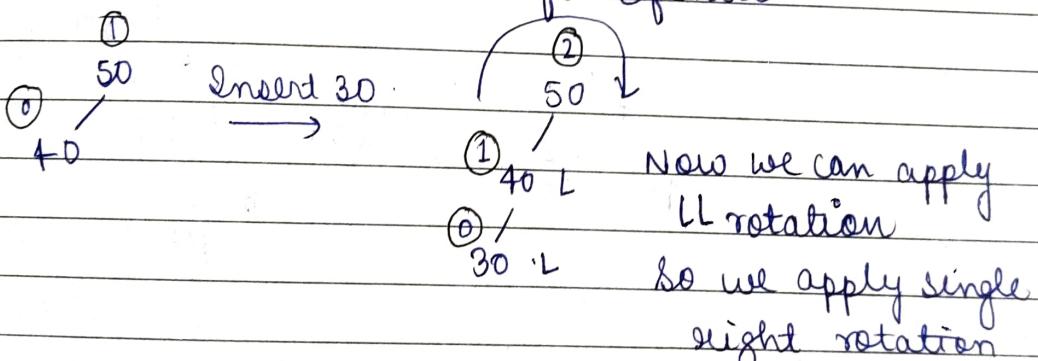


Add 22.

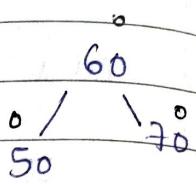
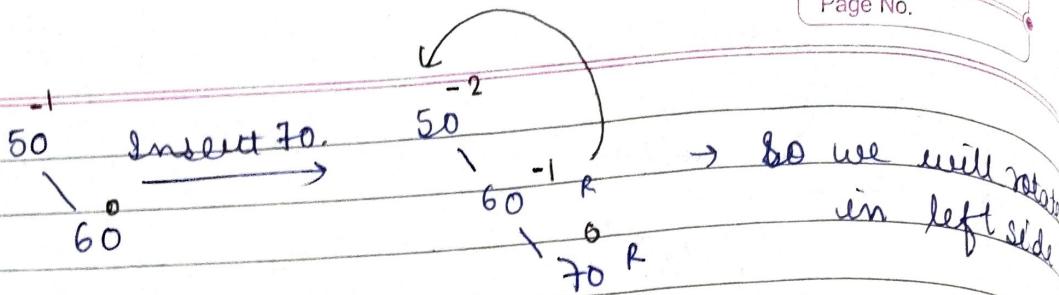


↔ we will use rotations
to amplify this case.

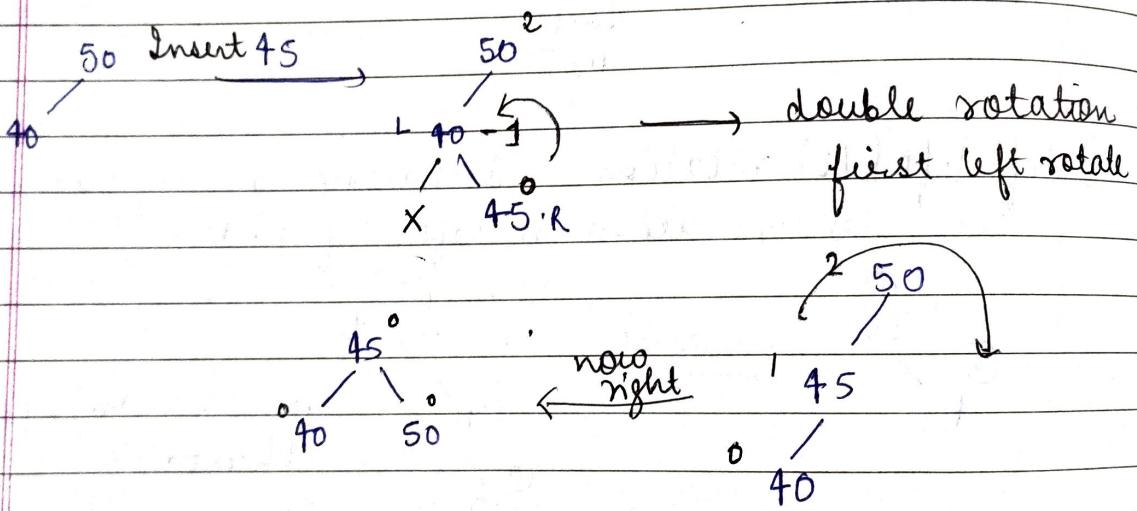
1) left left (LL) rotation: Imbalancing is caused due to insertion in left left node.



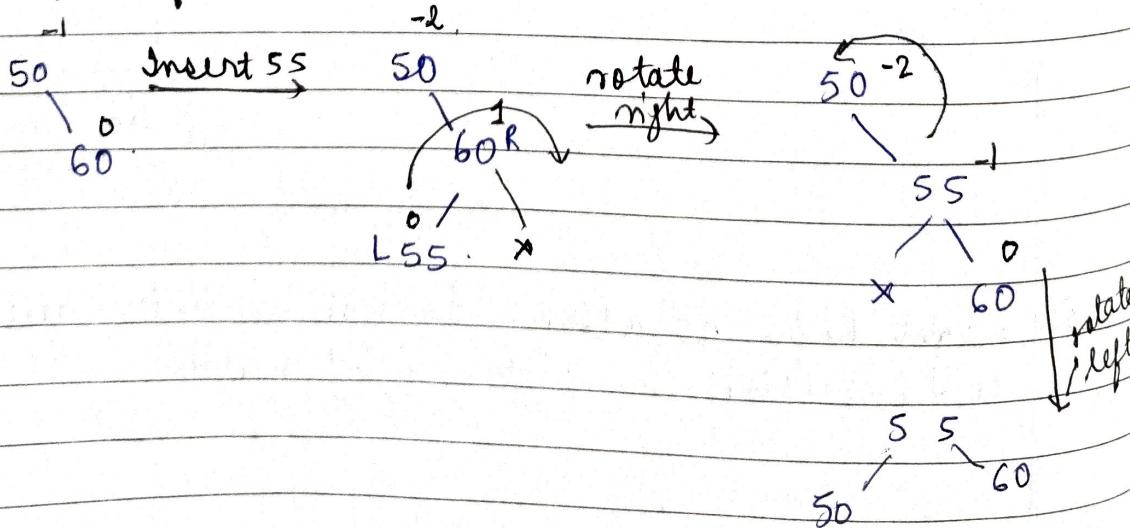
2) Right Right rotation: Imbalancing is caused due to insertion in right right node.



3) Left-Right rotation: Imbalancing caused due to insertion in left then right of the node.



4) Right-left rotation.

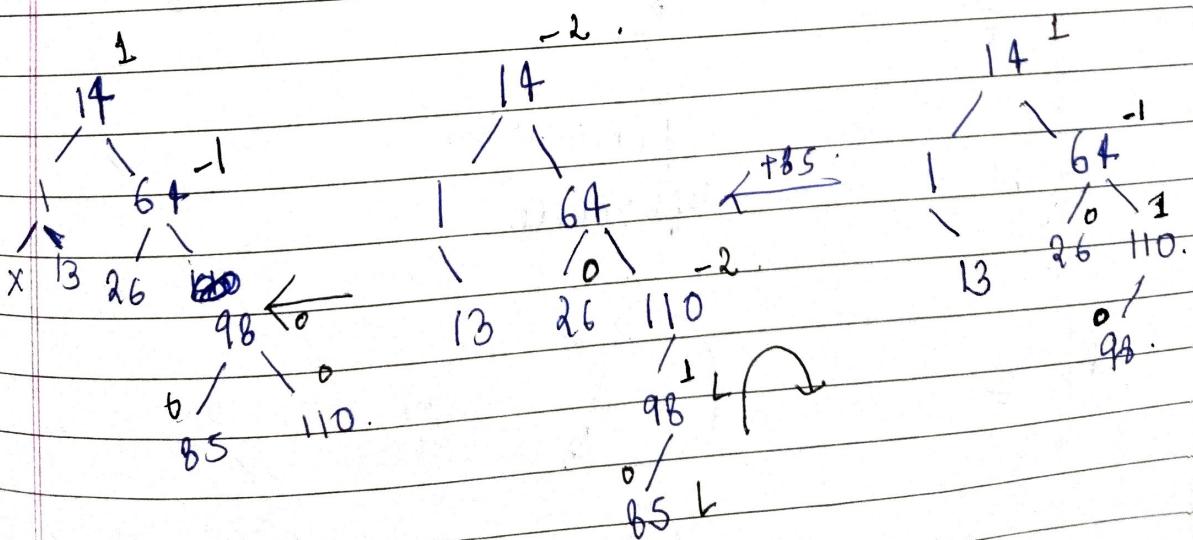
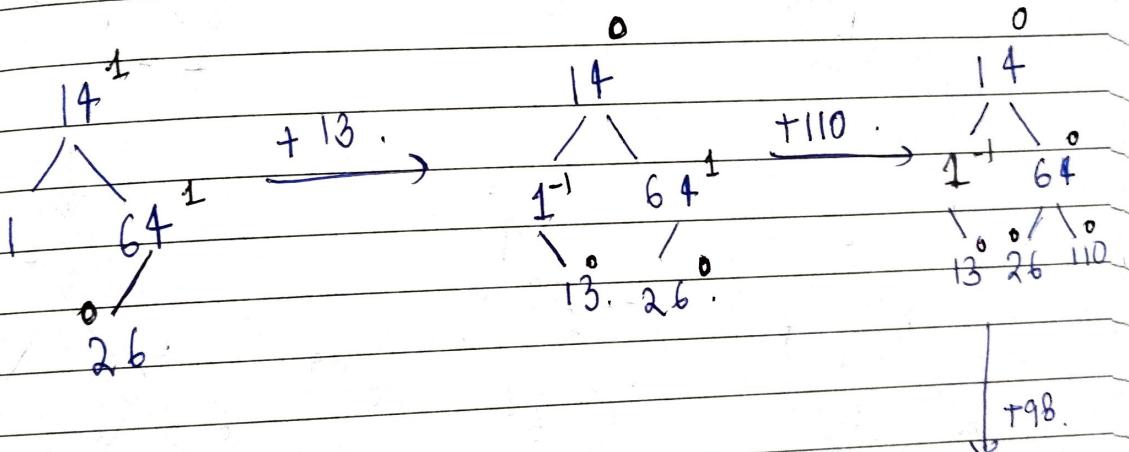
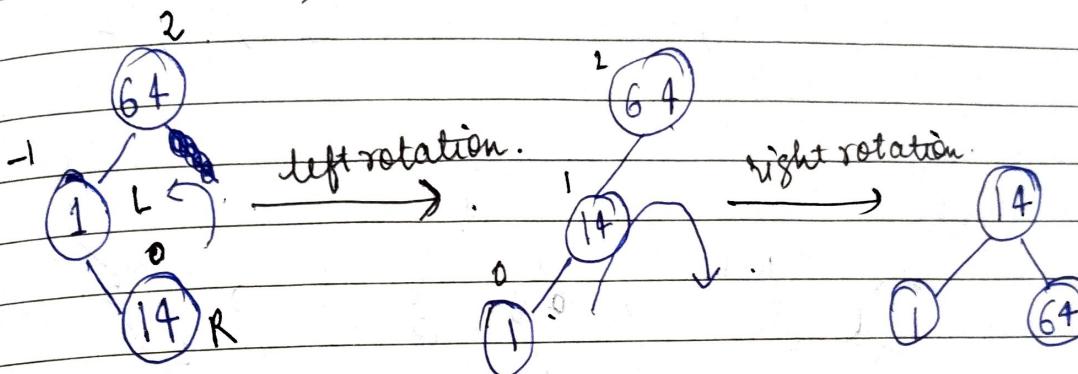


78

AVL tree insertion

Construct AVL search tree by inserting the following element.

64, 1, 14, 26, 13, 110, 98, 85.

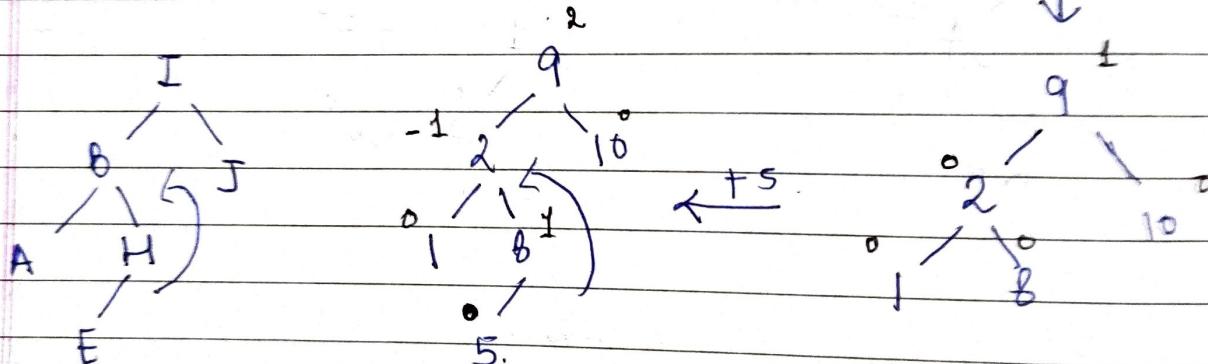
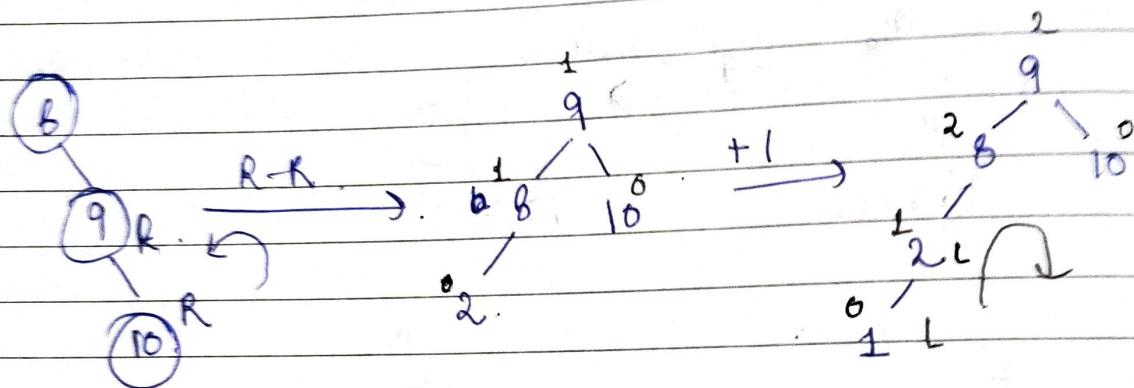


Q) Construct AVL tree having following keys.

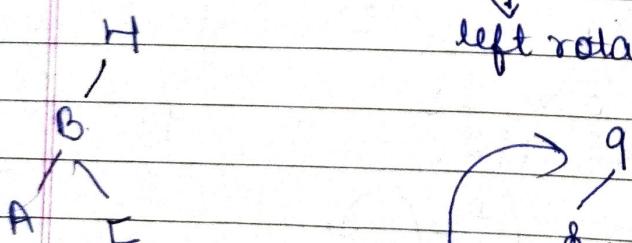
H, I, J, B, A, E, C, F, D, G, K, L

Sorted order: A B C D E F G H I J K L
1 2 3 4 5 6 7 8 9 10 11 12

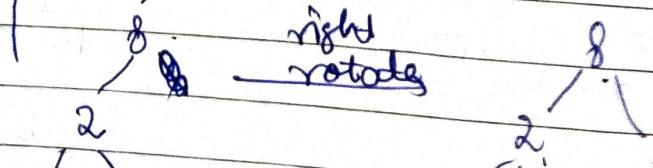
b, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12

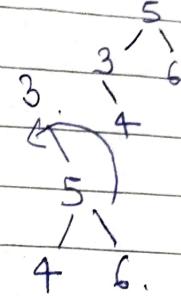
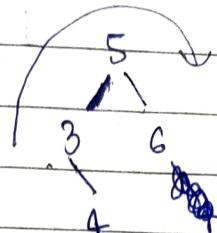
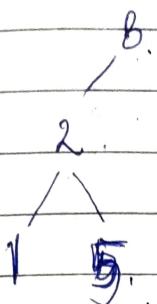
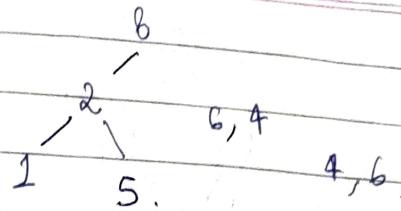
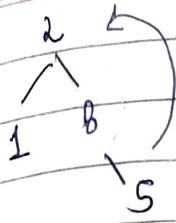


5.
LR rotation.
left rotate

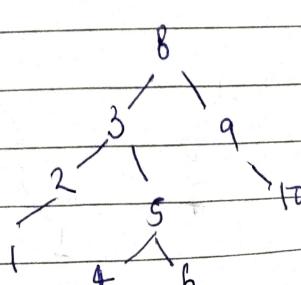
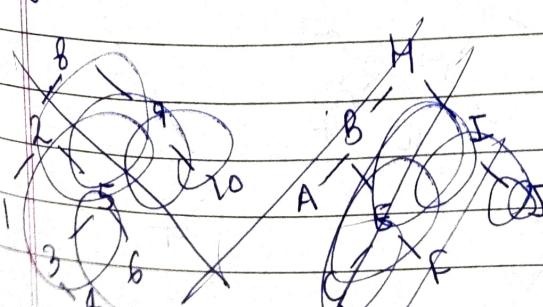
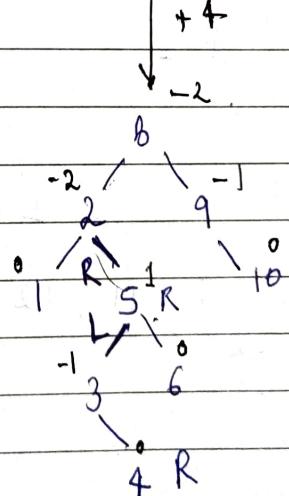
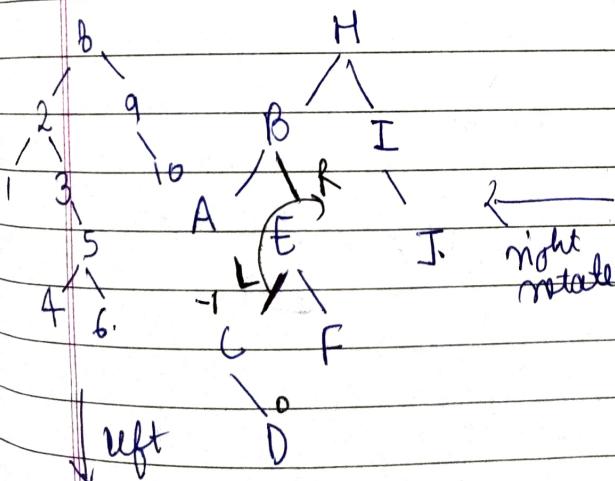
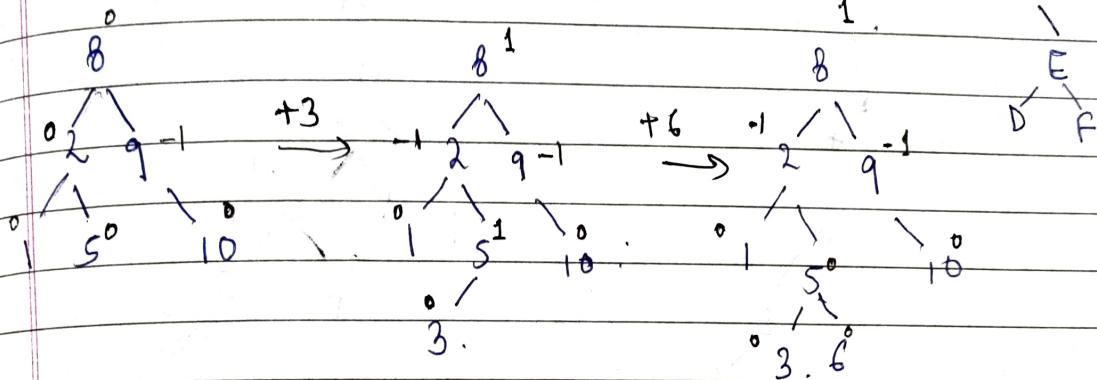


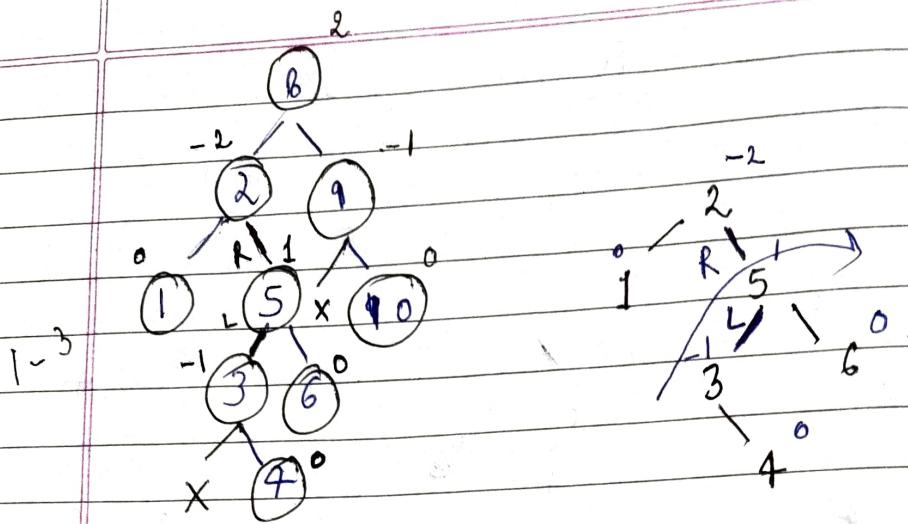
right
rotate





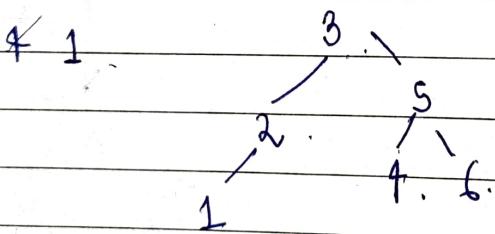
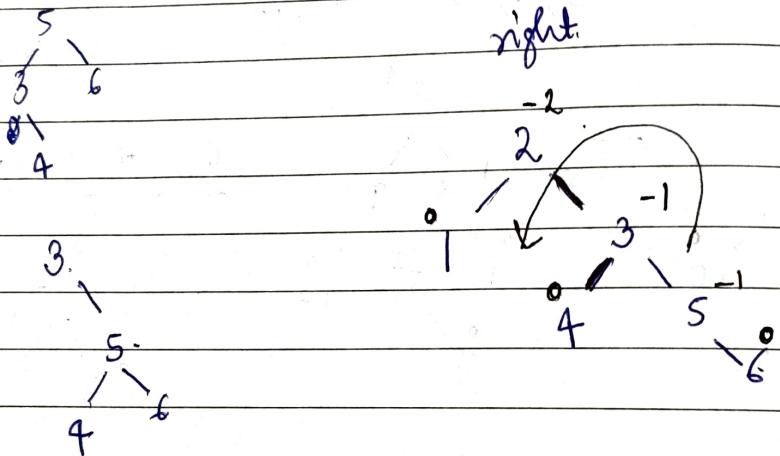
3, 6, +, 7, 11, 12





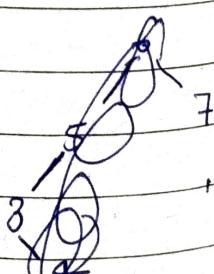
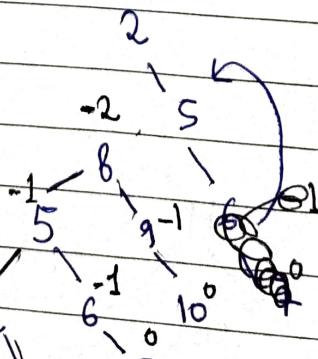
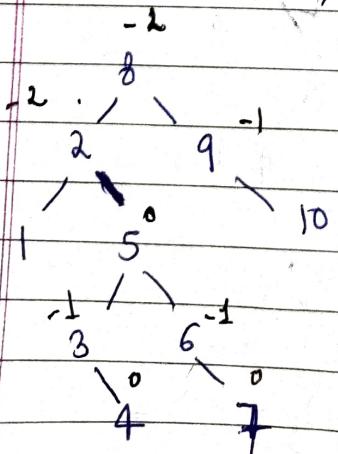
↓

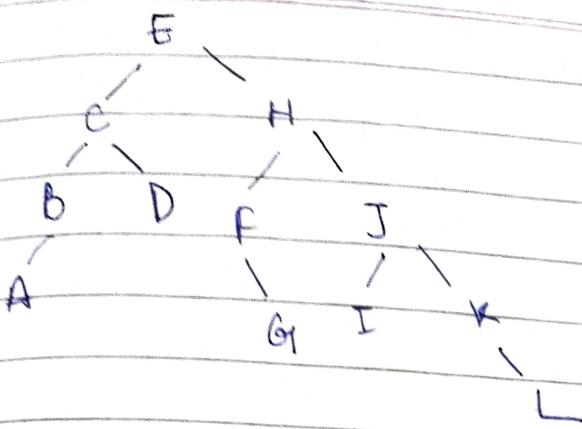
right.



7, 11, 12

2, Y, \$, X, /, 4, X





AVL tree deletion

(BB)

Deletion in AVL tree is similar as binary search tree. After deletion we restructured the tree if needed to maintain it right.

Step 1: Find element in the tree

Step 2: Delete the nodes as BST rule

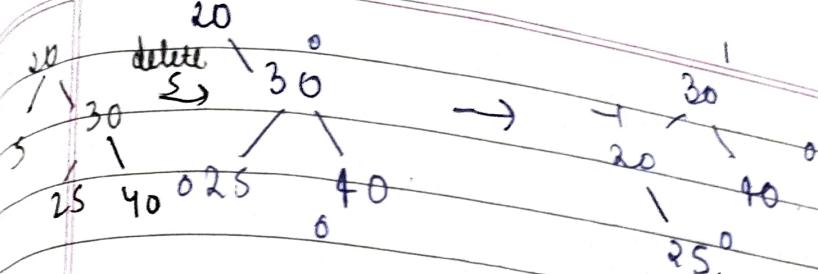
Step 3: Two case are possible if tree is unbalanced

Case 1: Deletion from right sub-tree

- i) if $BF = +2$ and $BF(N \rightarrow LC) = +1$, do LL rotation.
- ii) if $BF(N) = +2$ and $BF(N \rightarrow LC) = -1$, do LR rotation.
- iii) if $BF(N) = +2$ and $BF(N \rightarrow LC) = 0$, do L2 rotation

Case 2: Deletion from left sub-tree

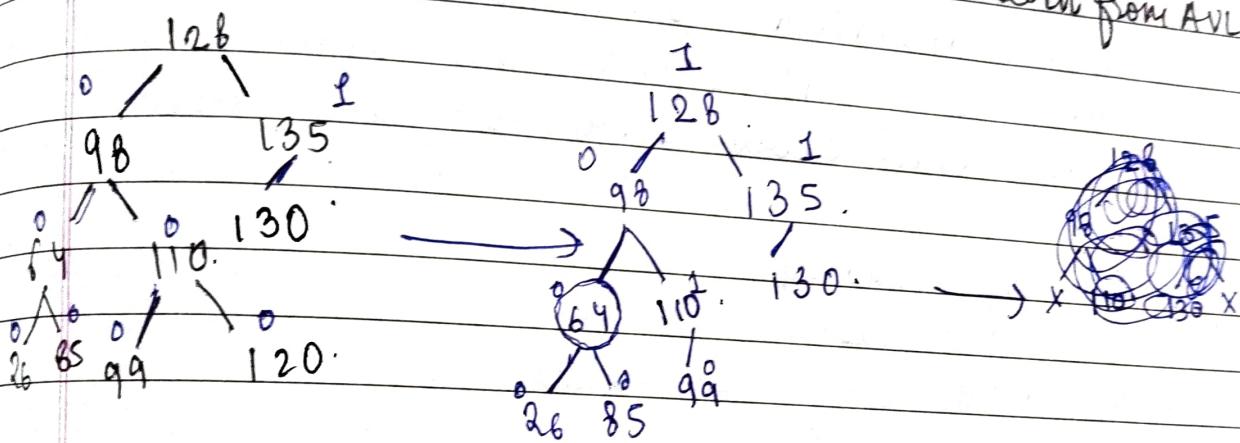
- i) if $BF(N) = -2$ and $BF(N \rightarrow RC) = -1$, then RR rot^n.
- ii) if $BF(N) = -2$ & $BF(N \rightarrow RC) = +1$, do RL rot^n.
- iii) if $BF(N) = -2$ & $BF(N \rightarrow RC) = 0$, do RR rot^n.



(b1)

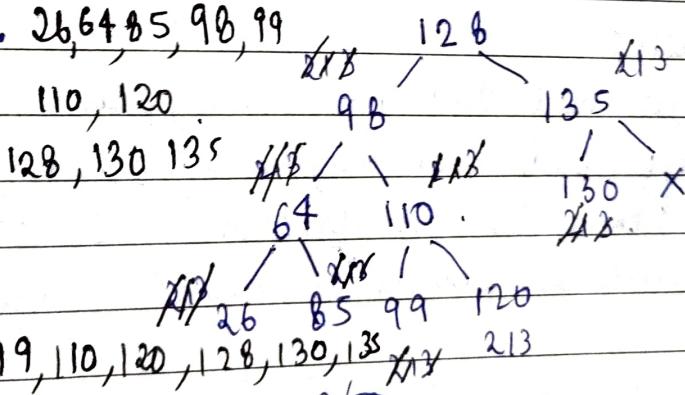
* Example of AVL tree deletion

Delete 120, 64, 130, 98, 128 in order from AVL.

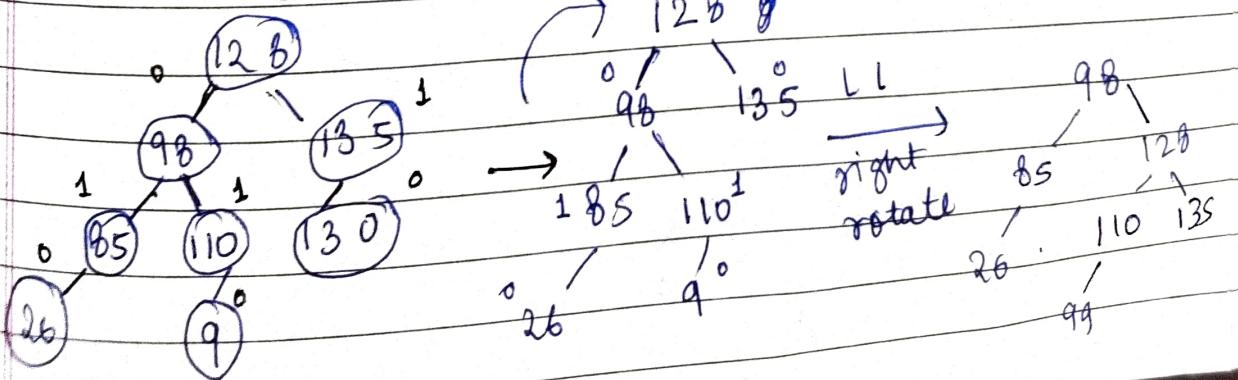


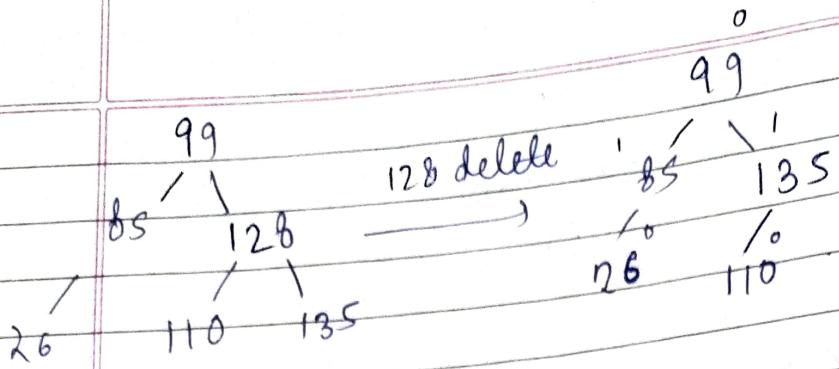
X/X

Inorder: 26, 64, 85, 98, 99



26, 64, 85, 98, 99, 110, 120, 128, 130, 135





(82) Threaded Binary Tree:

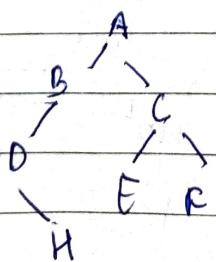
AJ Paeslis and C Thornton have proposed a new binary tree called "Threaded Binary tree", which make use of NULL pointer by references of other node. These extra references are called "Threads".

- 1) One way threading
- 2) Two way threading
- 3) two way threading with header node.

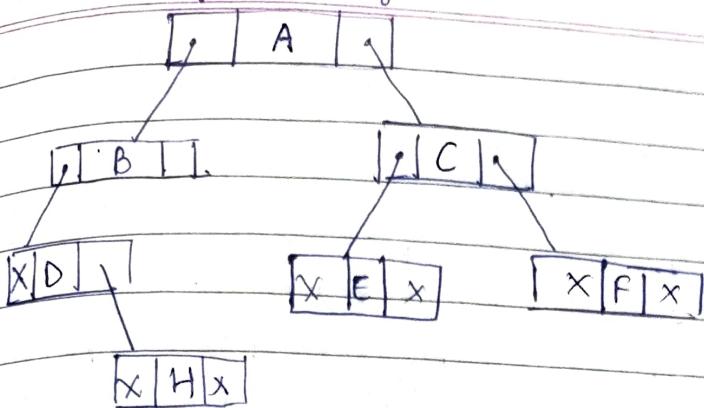
Cases:

- 1) Right child that are NULL, points to Inorder Successor
- 2) Left child that are NULL, points to Inorder Predecessor
- 3) If there is no inorder successor or predecessor then NULL points to header node.

In Binary tree in linked list there is many blockage.



left Data right.



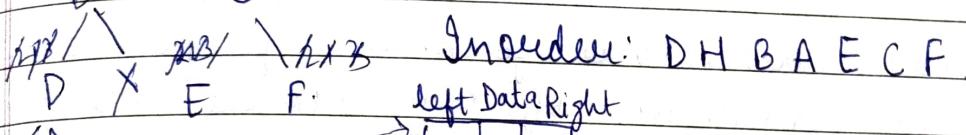
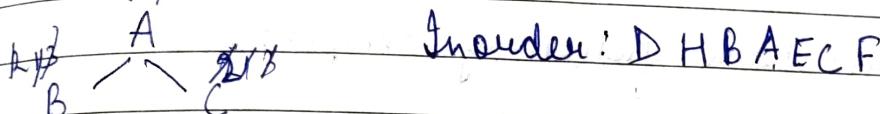
$$N = 7$$

$$2N = 14$$

$$N+1 \text{ unused} = 8$$

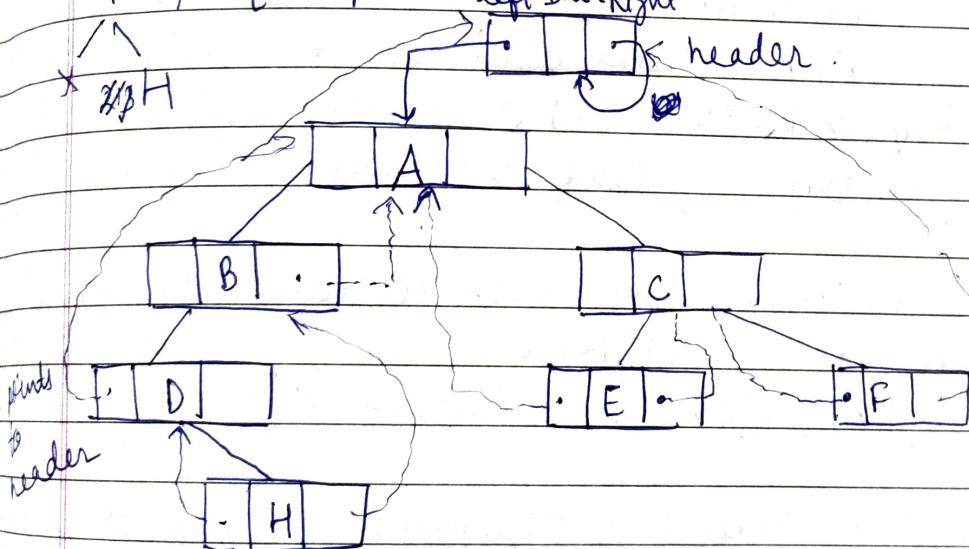
~~header~~

reference.



left Data Right

header

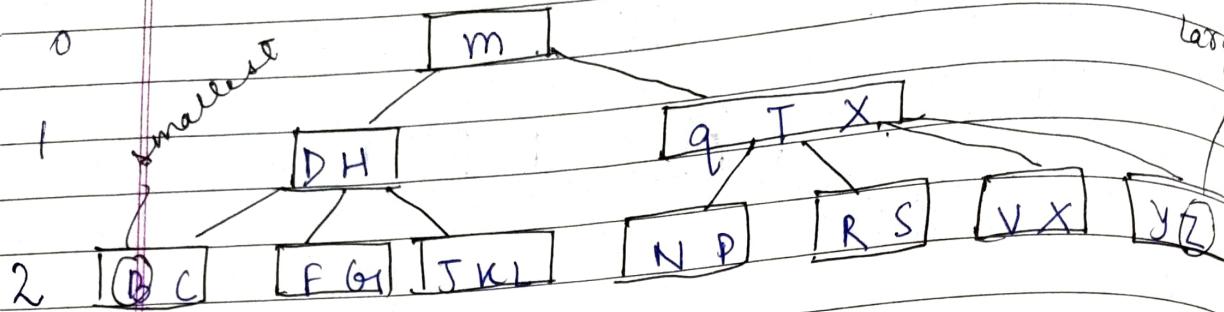


83

B-tree

B-tree are balanced search tree designed to work well on magnetic disk or secondary storage devices. B-tree node may have children from handful.

do thousand.



If internal node X contain $n[X]$ key then it has $n[X] + 1$ child and all leaves are at same depth.

B-tree have following Properties

- 1) Every node X has following properties
 - a) $n[X]$ numbers of key stored at X node.
 - b) the $n[X]$ key stored in non decreasing order
 $\text{key}_1[x] \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}$
 - c) leaf $x = \text{true}$ if leaf otherwise "False".
- 2) Every internal node also contain $n[X] + 1$ pointer to children.
 $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$.
- 3) The keys $\text{key}_i[x]$ separate the range of keys stored in subtrees.
 $k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \dots k_n[x] \leq k_{n+1}[x]$
- 4) All leaves has same depth which is height of tree.

5. There are some lower and upper bound on the no. of key a node contain. The Bound can be expressed as $t \geq 2$ called min. degree.

lower Bound: every node other than root contain all leaves $t-1$ key and t children.

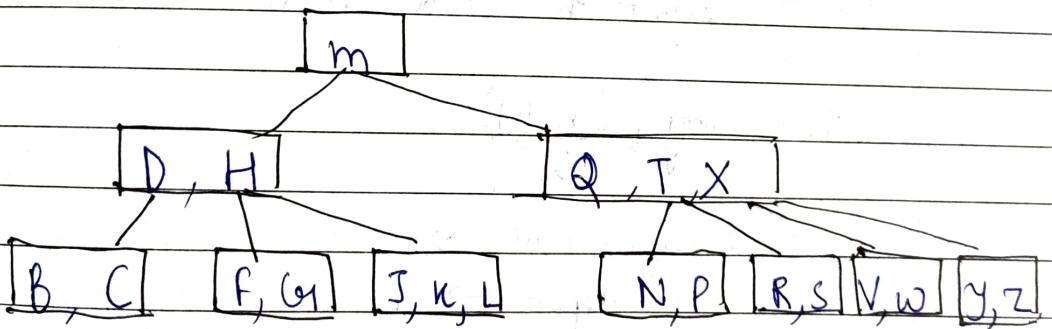
Upper bound: Every node contain at most $st-1$ key and st children.

Note: when $t=2$ every inter node has either 2, 3 or 4 children. called 2-3-4-tree.

(84)

B tree Operations.

- 1) B-tree search
- 2) B-tree create
- 3) B-tree insert
- 4) B-tree delete



B tree search algorithm

B-tree Search (x, k)

```
1. i ← 1
2. while  $i \leq n[x] \text{ and } k > \text{key}_i[x]$ 
3.   do  $i \leftarrow i + 1$ 
4. if  $i \leq n[x] \text{ and } k = \text{key}_i[x]$ 
5.   then return [ $x, i$ ]
6. if leaf [ $x$ ]
7.   then return NIL
8. else DISK-READ ( $c_i[x]$ )
9.   return B-tree search ( $c_i[x], k$ )
```

* Root of B-tree always in main memory
(Disk read never required)

B-tree

Theorem : If $n \geq 1$ then for any n -key B-tree (T) of height h and min^m. degree $t \geq 2$

$$h \leq \log_t \left(\frac{n+1}{2} \right)$$

no of key on root $\rightarrow 1$

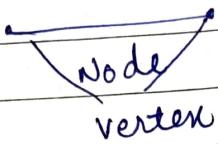
depth $\rightarrow 1$

(90)

Graph in Datastructure

A Graph can be defined as group of vertices & edges that are used to connect these vertices

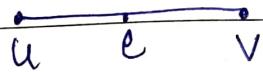
A graph consist of 2 things $G_1 = (V, E)$



V - set of vertex / element called node

$E \rightarrow$ set of edges & it each edge in E is unique
pair of vertices $[u, v]$

$$e = [u, v]$$

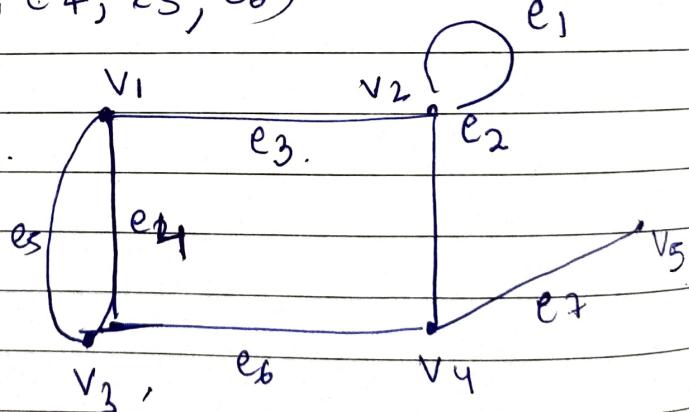


$$G_1 = V, E$$

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$$

Tree is always graph.
All graph is not tree



tree graph tree graph graph

i) Adjacent node

$$e_2 = \{v_2, v_4\}$$

v_2 is adjacent to v_4
 v_4 is adjacent to v_2

ii) Degree of node: No of edges connected to a node.

$$\deg(v_1) = 3$$

$$\deg(v_4) = 3$$

$$\deg(v_5) = 1$$

iii) Isolated node: Any node having degree as 0.

iv) Path: Route followed from one vertex to other.

$$v_1 \rightarrow v_5 \Rightarrow v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$$

v) Cycle: Starting and ending at same node

$$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_1$$

vi) Loop: If an edge have same start & end vertex

$$e_1 = \{v_2, v_2\}$$

degree is 2.

$$\deg(v_2) = 4$$

- Every tree is a graph but vice versa not true
 2) No cycle is in a tree.

Date: / /
 Page No. SPS

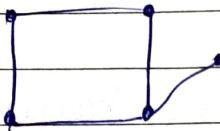
Parallel - Two edges having same vertex pair.

$$e_2, e_5 = \{v_1, v_3\}$$

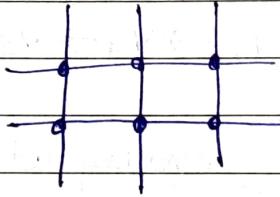
(Q1)

Types of Graph.

1) Finite graph \rightarrow No of edges & vertices are countable



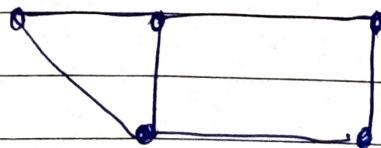
2) Infinite graph \rightarrow No of edges & vertices are not countable



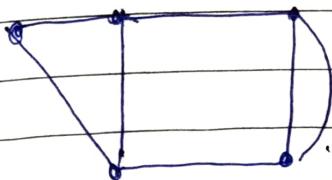
3) Trivial graph \rightarrow Single node with no edges.

degree = 0

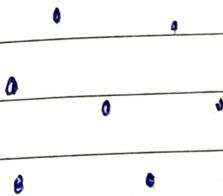
4) Simple graph \rightarrow Graph having no parallel edge or self loop.



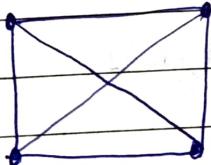
5. Multigraph - A graph having parallel edge but no self loop.



6. Null graph - No edges only vertex. (more than one node)



7. Complete graph - Every node has a degree of $n-1$



$$n-1 = 3.$$

$$\text{no of edges} = \frac{n(n-1)}{2}.$$

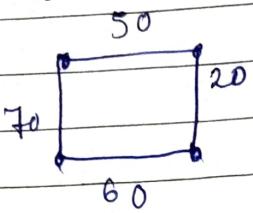
8. Pseudo graph - A graph having self loop as well as parallel edges



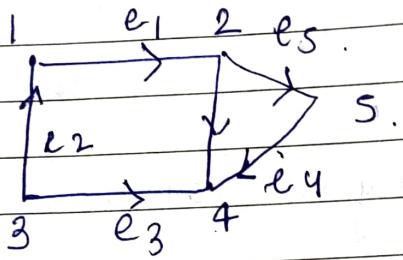
9. Regular graph - Every node has a same degree.



10. labelled graph: when we assign the edges with any weight or data. It is called labelled graph.



11. directed graph: A graph having directions in edges.

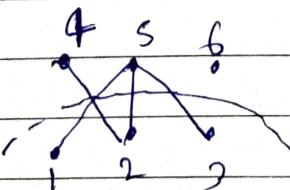


Degree here are of 2 types.

for node

2 Indegree - 1
Out degree - 2.

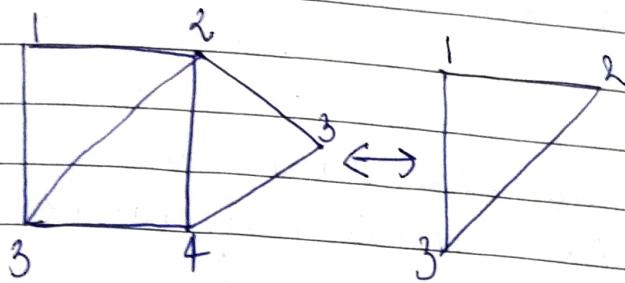
12. Bipartite graph: when a graph is divided into 2 parts such that each edge has one of its ends in both.



$$V' = \{1, 2, 3\}$$

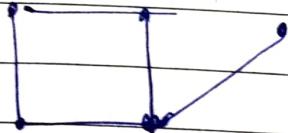
$$V'' = \{4, 5, 6\}$$

13. Sub-graph - A part of graph

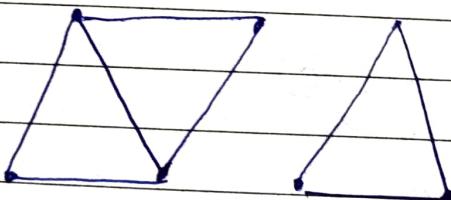


14. Connected / disconnected graph

when a graph is connected in edges all there is path to reach them.



When a graph are in space & not connected with each other.



(92)

Graph representation

- 1) Sequential Representation (2D Array)
- 2) linked list Representation (linked list)

Sequential representation is achieved by adjacency matrix. In this representation we have to construct $n \times n$ matrix where n is number of vertices.

If there is edge from vertex i to j then corresponding element of matrix:

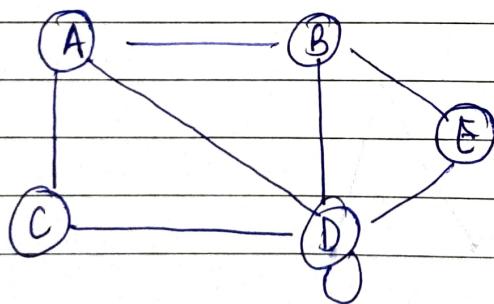
$$A_{ij} = 1 \text{ otherwise } A_{ij} = 0$$

or

$$A_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise} \end{cases}$$

Types of graph

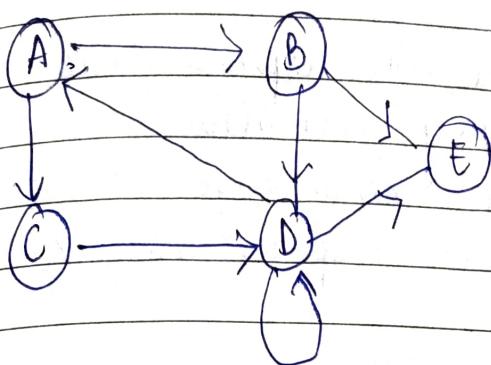
1) undirected graph



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

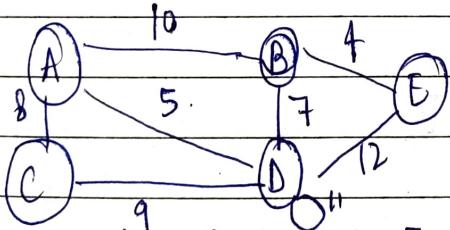
adjacency matrix

ii) directed graph



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

iii) Undirected weighted graph



	A	B	C	D	E
A	0	10	8	5	0
B	10	0	0	7	4
C	8	0	0	9	0
D	5	7	9	11	12
E	0	4	0	12	0

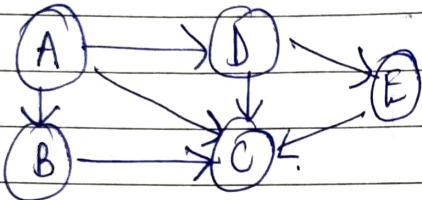
(93)

Linked list Representation of graph

Problems in Sequential representation

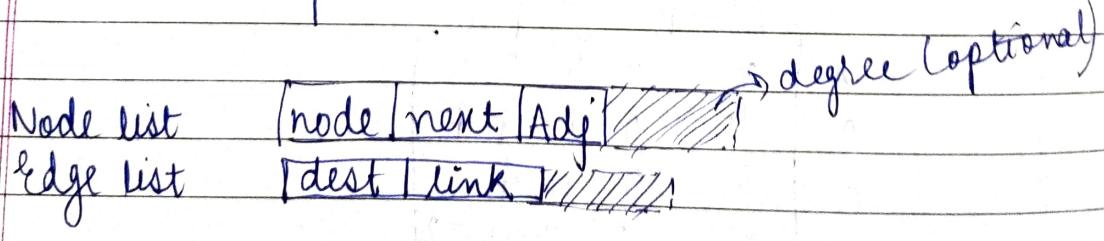
- 1) No dynamic memory allocation
- 2) When no of node = no of edges there will be more no of zeroes (sparse matrix)

So we use linked list



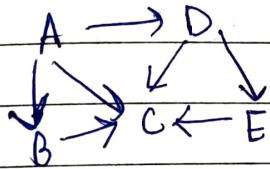
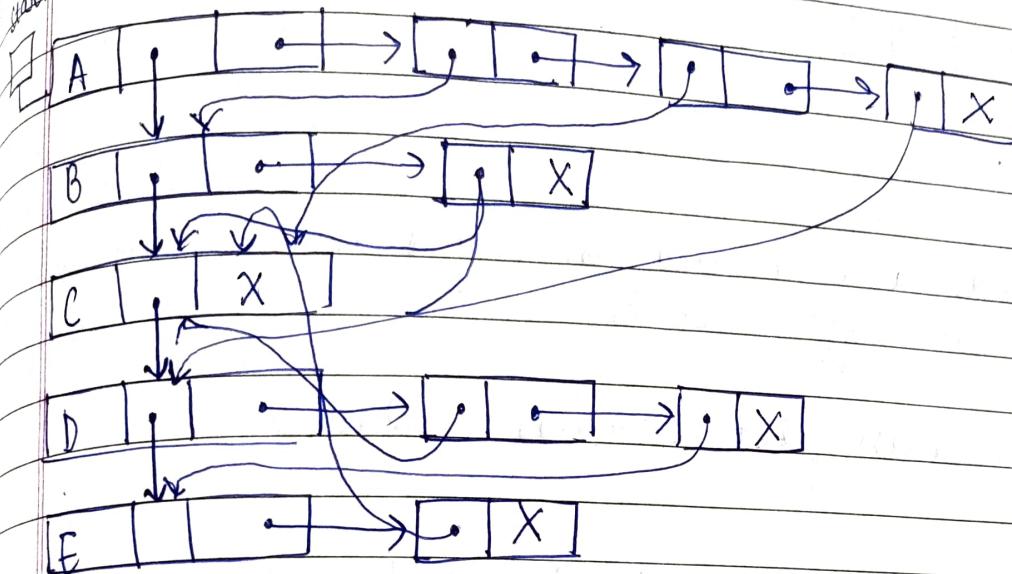
Adjacency list of graph.

node	Adjacency list
A	D, C, B
B	C
C	—
D	C, E
E	C



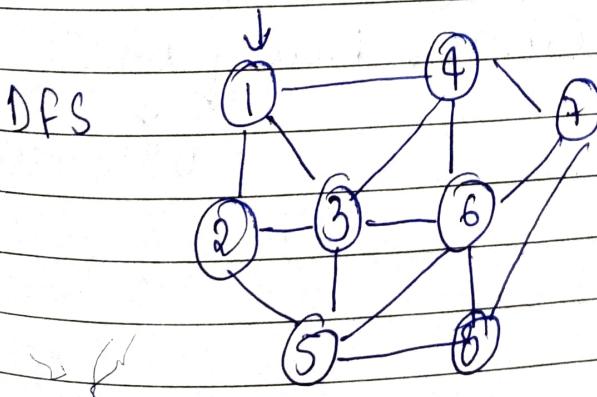
Node list

Node next adj



(94)
Graph traversing

- 1) Depth first search (DFS) uses stack (LIFO)
- 2) Breadth first search (BFS) uses queue (FIFO)



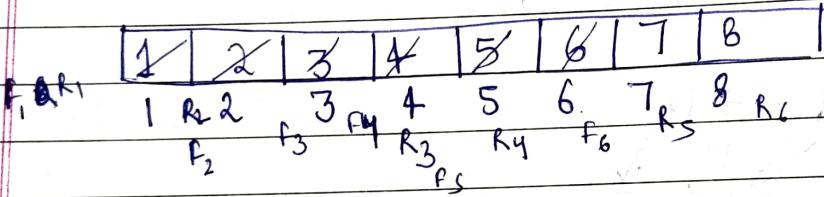
	8
	7
	6
	5
4	8 7 5
3	4 6
2	3
1	1 2

Op: 1

visited: 1 4 7 8 5 6 3 2

Node can be inserted in any order so we can have diffⁿ results.

2. Breadth first search (Queue)

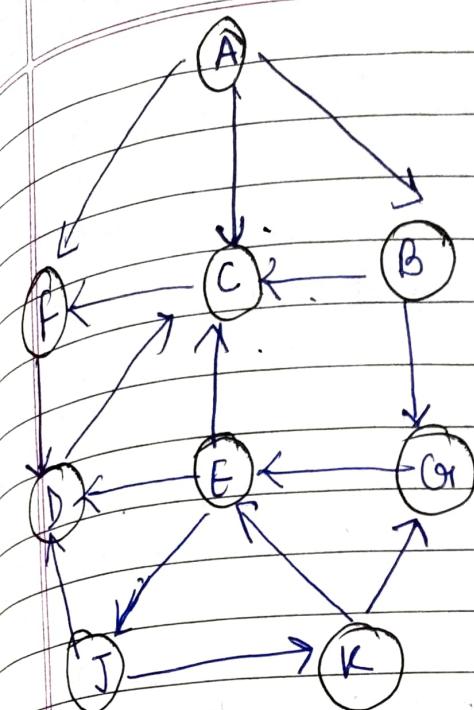


Visited: 1, 2, 3, 4, 5, 6, 7, 8
4, 7, 6, 3, 1, 8, 5, 2

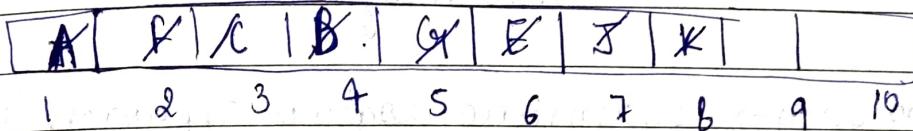
(95)

DFS | BFS solved

- ① find min^m path from (A to J) (BFS) \rightarrow queue
 - ② Point all the reachable Node from J
- ↓
- DFS \rightarrow stack



Node	Adjacency list
1. A	F, C, B
2. B	C, G
3. C	F
4. D	C
5. E	D, C, J
6. F	D
7. G	E, F
8. H	D, K
9. I	E, G



visited : A F C B D G E J ^{stop} K

Parent : φ A A A B f B G E

J ← E ← G ← B ← A shortest path

Visited	Parent
A	φ
F	A
C	A
B	A.
D	f
G	B
E	G
J	E
K	J

9	
8	
7	
6	
5	
4	
3	C D G F
2	E D K J
1	J S

Visited : J K G C F E D

Reachable

nodes: J, K, G, C, F, E, D

(1)

Difference between Algorithm, Pseudocode
and Program

Algorithm → Systematic logical approach to solve any problem. It is written in Natural language.

Pseudocode → It is simple version of programming code that doesn't require any strict programming language syntax.

Program → It is exact code in any particular programming language.

Let's take example of linear search

Algorithm

Start from left element of arr[] and one by one compare x with each element of arr[]. If x match return index of element else return -1

Pseudocode

```
function LSearch(list, x)
    for index ← 0 to length(list)
        if list[index] == x then
            return index
    end if
end loop.
return -1.
End
```

Program

```
int LSearch(int a[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (a[i] == x)
            return i;
    return -1;
}
```

(2)

Introduction to Recurrence Rel^n

when an algorithm contain a recursive call to itself its running time can be described by Recurrence rel^n.

eg: Recurrence rel^n of merge sort

$$T(n) \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases}$$

1) void fun(int n) = T_n

{

if ($n > 0$) — 1

{

print (n) — 1

fun(n-1) = $T(n-1)$

}

$$T(n) = 1 + 1 + T(n-1)$$

$$T(n) = 2 + T(n-1)$$

$$T(n) = C + T(n-1)$$

$n = 0$

$$T(0) = C + T(0)$$

1 if $n = 0$

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) & \text{if } n > 0. \end{cases}$$

2)

void ACn) — T_n

{

if ($n > 0$) — 1

{

for (i=0; i<n; i++) — $n+1$

print(n) — n

ACn-1) — $T(n-1)$

{

$$T(n) = 1 + n + n + T(n-1)$$

$$= T(n-1) + C + Cn = T(n-1) + n$$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+n & n>0 \end{cases}$$

③ fib(m)

if ($n \leq 1$)
return 1

else

return fib(m-1) + fib(m-2) — $T(n-1)+T(n-2)$

$$T(n) = T(n-1) + T(n-2) + 2$$

+1

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1)+T(n-2) & n > 2 \end{cases}$$

③

Solving Recurrence Relation

There are 3 method to solve Recurrence

- ① Substitution method
- ② Recursion tree method
- ③ Master method.

① Substitution method.

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1)+1 & \text{if } n>0 \end{cases}$$

Substitution method

forward substitution back substitution

$$T(n) = T(n-1) + 1$$

$$n = n-1$$

$$T(n-1) = T(n-2) + 1$$

$$n = n-2$$

$$T(n-2) = T(n-3) + 1 + 1$$

$$T(n) = T(n-2) + 1 + 1$$

$$T(n) = T(n-2) + 2$$

$$T(n) = T(n-3) + 3$$

!

$$= T(n-k) + k$$

$$n - k = 0$$

$$= T(0) + n$$

$$= 1 + n$$

$$T(n) = O(n)$$

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

$$T(n) = T(n-1) + n$$

$$n = n-1$$

$$T(n-1) = T(n-2) + n-1$$

$$T(n) = T(n-2) + n-1 + n$$

$$T(n) = T(n-2) + 2n - 1$$

$$T(n) = 2T(n-k) + kn$$

$$n-k=0$$

$$n=k$$

$$\begin{aligned} T(n) &= T(0) + n^2 \\ &= 1 + n^2 \\ &= O(n^2) \end{aligned}$$

$$3) T(n) = \begin{cases} 1 & n=1 \\ 2T\left(\frac{n}{2}\right) + n & n>1 \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$n = \frac{n}{2}$$

$$n = \frac{n}{4}$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$

~~$$T(n) = 2 \times \left[2T\left(\frac{n}{8}\right) + \frac{n}{4} \right] + \frac{n}{2}$$~~

~~$$= 2 \times \left[2T\left(\frac{n}{16}\right) + \frac{n}{8} \right] + \frac{n}{4}$$~~

\ddots

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2 \left[2T\left(\frac{n}{4}\right) + \frac{n}{2} \right] + n$$

$$\geq 2^2 T\left(\frac{n}{8}\right) + 2n$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log n = k \log 2$$

$$k = \log n$$

$$= 2^k T(1) + kn$$

$$= 2^{\log n} + kn$$

$$= n + n \log n$$

$$= O(n \log n)$$

Recursion tree method

$$Q: T(n) = \begin{cases} 1 & \text{if, } n=0 \\ T(n-1)+n, & \text{if, } n>0 \end{cases}$$

Recurrence Relations

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + n & \text{if } n>0 \end{cases}$$

$$T(n) = n$$

|

$$T(n-1) = (n-1)$$

|

$$T(n-2) = n-2$$

|

$$T(n-3) = n-3$$

:

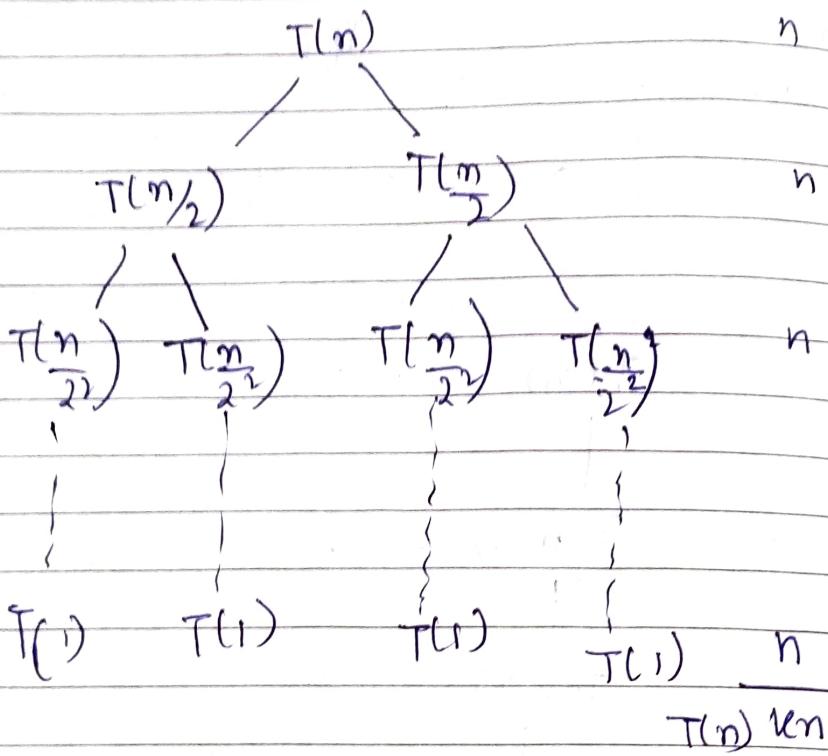
$$T(0) = 1$$

$$= 1 + 2 + \dots + (n-3) + (n-2) + (n-1) + n$$

$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

$$= O(n^2)$$

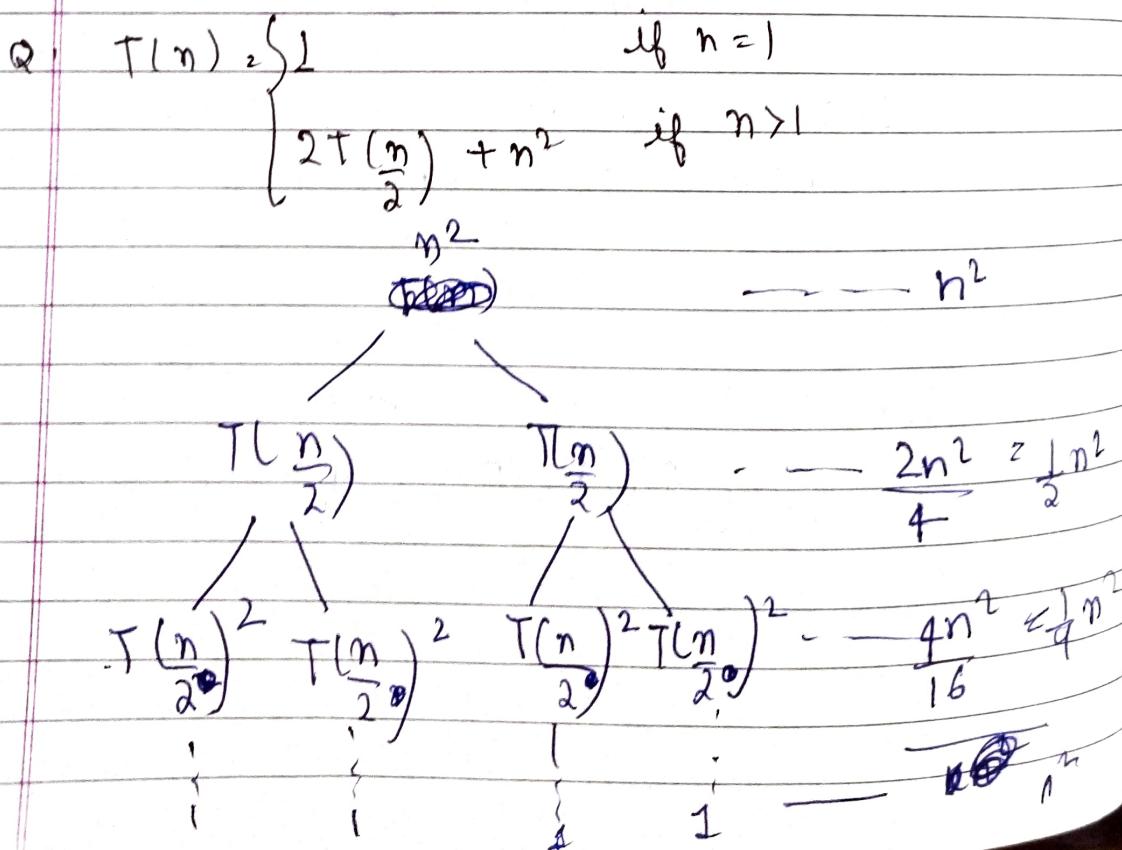
a) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n>1 \end{cases}$



$$\frac{n}{2^k} = 1$$

$$k = \log n$$

$$T(n) = n \log n$$



$$\frac{n}{2^n} > 1$$

$$n > \log n$$

$$T(n) = n^2 n$$

$$\approx n^2 \log n$$

$$n^2 + \frac{n^2}{4} + \frac{n^2}{8} + \dots + 1$$

$$= n^2 \left[1 \times \left(\frac{1}{1-\frac{1}{2}} \right) \right]$$

$$T(n) \approx 2n^2$$

$$T(n) = O(n^2)$$

Master Method (Divide & conquer)

The problem is divided into number of subproblems each of size $\frac{n}{b}$ and need time $f(n)$ to combine the solution. Then the running time $T(n)$ can be:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

where $a > 1$,
 $b > 1$

$f(n)$ is asymptotically positive function
 $\frac{n}{b}$ means $\left[\frac{n}{b}\right]$ see

$$\left[\frac{n}{b}\right]$$

$T(n)$ can be bounded asymptotically as follows

1) $f(n) < n^{\log_b a}$ or $f(n) = O(n^{\log_b a} - b)$ for some constant $b > 0$ then $T(n) = O(n^{\log_b a})$

2) $f(n) = n^{\log_b a}$ or $f(n) = O(n^{\log_b a})$ then $T(n) = (n^{\log_b a} \log n)$

3) $f(n) > n^{\log_b a}$ or $f(n) = \Omega(n^{\log_b a} + b)$ for some const $p > 0$ and if $a(f(\frac{n}{b})) \leq c f(n)$ for some constant $c < 1$ & all sufficiently large n , then $T(n) = O(f(n))$

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$aT\left(\frac{n}{b}\right) + f(n)$$

$$a = 9 \quad b = 3$$

$$\begin{aligned} n^{\log_a b} &= n^{\log_3 9} \\ &= n^2 \\ f(n) &= n \quad n^{\log_a b} = n^2 \end{aligned}$$

$$f(n) < n^2$$

$$\begin{aligned} \text{then } T(n) &= O(n^{\log_b a}) \\ &\in O(n^2) \end{aligned}$$

$$T(n) = T\left(\frac{n}{3}\right) + 1$$

$$a = 1$$

$$b = 3$$

$$n^{\log_b a} = n^{\log_3 1}$$

$$= n^0$$

$$= 1$$

$$f(n) = 1$$

$$f(n) = n^{\log_b a} = 1$$

$$T(n) = (n^{\log_b a}, \log n)$$

$$T(n) = (1 \log n)$$

$$T(n) = \log n$$

$$T(m) = 3T\left(\frac{m}{4}\right) + m \log m$$

$$a = 3 \quad b = 4$$

$$n^{\log_b a} = n^{\log_4 3}$$

$$f(n) = n \log n$$

$$T(m) = f(n)$$

$$f(n) > n \log n$$

$$\geq O(n \log n)$$

Solving Recurrences.

$$1) T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$\begin{aligned} a &= 1 & f(n) &= O(1) = 1 \\ b &= 2 \end{aligned}$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = n^{\log_b a}$$

$$T(n) = (n^{\log_b a} \times \log n)$$

$$= (n^0 \times \log n)$$

$$= \log n$$

$$2) T(n) = 2 T\left(\frac{n}{2}\right) + n^3$$

$$a = 2 \quad b = 2 \quad f(n) = n^3$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) > n^{\log_b a}$$

$$T(n) = O(f(n))$$

$$= O(n^3)$$

Sorting Algorithms

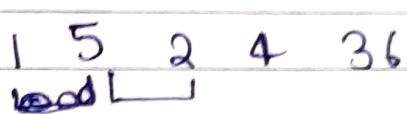
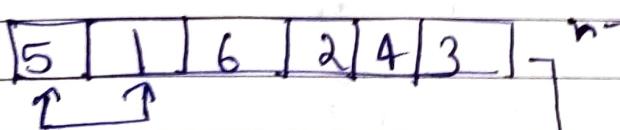
- 1) Bubble sort
- 2) Selection sort
- 3) Insertion sort
- 4) Shell sort
- 5) Quick sort
- 6) Merge sort
- 7) Heap sort

Bubble sort

Compare 2 adjacent elements a & b

If $a > b$ then swap a and b.

1	2	3	4	5	6
5	1	6	12	4	3.



12 4 3 5 6

1 5 6 2 4 3.

1 2 3 4 5 6.

1 5 2 6 4 3.

1 5 2 4 6 3.

1 5 2 4 3 6.

1 2 5 4 3 6

1 2 4 5 3 6

1 2 4 3 5 6

1 2 4 3 5 6.

Pass 1 $\rightarrow n-1$

Pass 2 $\rightarrow n-2$

Bubble sort (A, n)

1. ~~for (i = n; i > 1; i--)~~
2. ~~for (j = 1; j <= i - 1; j++)~~
3. ~~if a[j] > a[j + 1]~~
4. ~~swap(a[j], a[j + 1])~~
- 5.

Time complexity = $O(n^2)$.

$$\begin{aligned}
 T(n) &= n-1 + n-2 + n-3 + \dots + 1 \\
 &= \frac{n(n-1)}{2} \\
 &= \frac{n^2-n}{2} \\
 &\approx O(n^2)
 \end{aligned}$$

Time

Worst case = $O(n^2)$

Best case = $O(n)$

Avg case = $O(n^2)$

Space

$O(1)$ while swapping temp variable