Complete DSA Notes

— By Riti Kumari

1

Data structure

(meaningless) Raw facts & figure — Data

structure — organization of data

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.

eg: Array, linkedlist, stack, queue

Problem
|
datastructure ← Algorithm
|
Program → C/C++

Data → Raw facts
Information → Meaningful data

2

Types of DS

Data structure

Primitive DS
1) int
2) char
3) float
4) double
5) boolean

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	Non Linear DS
1) All elements are arranged in linear order where each element had successor and predecessor except first & last element.	This DS doesnt form a sequence. Data element are arranged hierarchical
2) Single level involved	2) Multilevel involved.
3) Used in s/w development	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 ~~of~~ 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

Analysis ↓ 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 → clear & unambiguious steps (b/o, a/o)
4) Finiteness → Algorithm should terminate after few steps
5) Effectiveness → (Time & space)

6

Analyzing Algorithms

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

Efficiency
↙ ↘
Amt. of computer memory | Amt. of computer time

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 " " " " "

SPS Date: / /
Page No.

(1) Sub(a, n)
{ — 0
s = 0 — 1
for i = 1 to n do → n+1
s = s + a[i] → n
return s — 1
}

$\cancel{2} + \cancel{2}n$

O(n)

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

for i ← 1 to m do. ⟶ m+1
for i ← 1 to p do ⟶ m(p+1)

c[i, j] = 0. ⟶ 1
for k ← 1 to n do. ⟶ mp(n+1)
c[i, j] ← c[i, j] + a[i, k] * b[k, j]
return c ↳ m·p n.

O(mpn)

(9)

Space complexity Analysis

Amount of memory needs to run to completion.

Space complexity S(P) → Constant space + Auxiliary Space

i/p, local variable — temp variable

Date: / /
Page No.

```
abc(a,b,c)
{
return a+b+b*c+(b+b-c)/a+b+4.0
}
```

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

→ $S(p) = O(1)$

2.
```
Sum(a,n)
{
  s = 0
  for i = 1 to n do
      s = s+a[i]
    return s;
}.
```

$$S(p) = (\underbrace{n*1}_{a} + \underbrace{1}_{s} + 1) + \underbrace{1}_{i}$$

$$= (n+2) + \cancel{1}$$

$$= O(1)$$

We don't take in account const. space.

3.
```
def Rsum(a,n)
    if n <= 0
      return 0
    else
      return a[-1] + Rsum(a[-2], n-1)
```

Date:
Page No.

$S(P)$ = no of stack frames * space per stack frame

$$S(P) = n * \underbrace{\text{size of}(a) + \text{size of}(n) * \text{size of } a(n)}_{\text{const}}$$

$$= O(n)$$

10

Asymptotic notations
(Growth of a function)

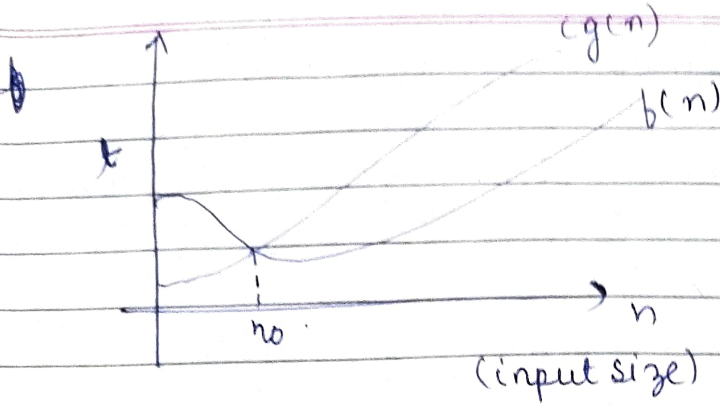
Growth of $func^n$.

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

Asymptotic Notations
1) Big-Oh (O) → upper Bound
2) Big Ohmega (Ω) → lower Bound
3) ~~Small~~ Theta (Θ) → Avg Bound. (type bound)
4) Small-oh (o)
5) Small omega (ω)

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

1) Big Oh (O) : The $func^n$ $f(n) = O\,g(n)$ if there exists positive constant c and n_0 such that $f(n) \leq c \cdot g(n)$ $\forall n,\ n \geq n_0$.



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

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

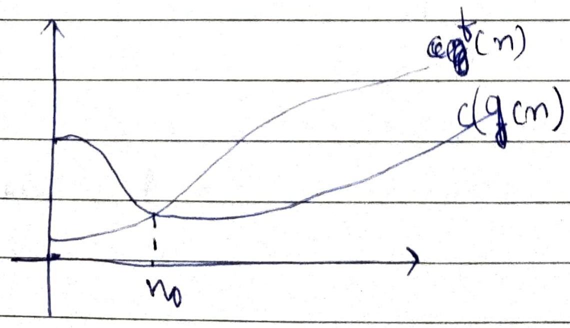
$3n+2 \leq c \cdot n$

$c = 1$ ×
$c = 2$ ×
$c = 3$ ×
$3n+2 \leq 4n$ $\quad c = 4$ ✓ $\quad n_0 = ?$

$n_0 \begin{cases} n = 1 \times \\ n = 2 ✓ \\ n = 3 ✓ \end{cases}$

$c = 4 \quad n_0 = 2$

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, $n \geq n_0$, $c > 0$



$f(n) \geq c g(n)$
$c > 0$
$n \geq n_0$
$n_0 > 0$
$f(n) = \Omega g(n)$

Date: / /
Page No.

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

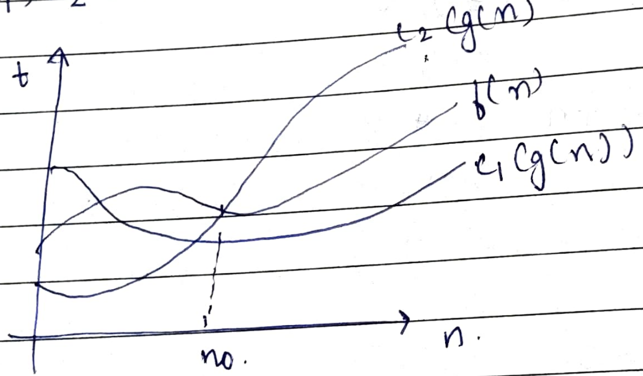
$f(n) \geq c \cdot g(n)$ $\quad c > 0$, $n_0 > 0$.

$3n+2 \geq 3n$

$c = 3$ ✓
$n = 1$ ✓
$n = 2$ ✓

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

3. Theta (θ): The fun. $f(n) = \theta(n)$ if there exists a +ve constant c_1, c_2, and n_0 such that
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
$c_1, c_2 > 0$.



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

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

$c_1 = 3$ $\quad c_2 = 4$
$n \geq 1$ $\quad n \geq 2$

$f(n) = \theta(n)$ for $c_1 = 3$, $c_2 = 4$, $n \geq 2$

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 ;

10

a

1000 1001

address of a

Element - item/data stored in array

index - location of element from 0^{th} - $n-1^{th}$

Address - numerical value of 1^{st} byte at which item is stored

int a[5];

0	1	2	3	4
10	15	16	17	18
1000	1004	1008	1012	. . .

base address.

10	11	12	13	14

1000

a[0] = 10
a[1] = 11
a[2] = 12
a[3] = 13
a[4] = 14

base address = 1000

address of a[i] = base address + index * size of element.

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

How to find location of any array element

loc A [i] = b + w (i - l)

i → element whose address to be found
B = base address
w = size of element
l = lower bound if not given take 0.

loc A[2] = 1000 + 2 (2 - 0)
= 1000 + 4
= 1004

A[1 - 100] b = 1000 w = 4 bytes

$A[50] = 1000 + 4(50-1)$
$= 1196$

Q. $A[-50 \ldots 50]$ $ba = 999$ $w = 10$.
l = −50, u = 50

$A[49] = 999 + 10 \times (49 - (-50))$
$= 1989$

(13)

Two dimensional Array

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

~~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 represnt^n

	a_{00} 0	a_{01}	a_{02} 2
0	10	20	3
1	40 a_{10}	50 a_{11}	60 a_{12}

10	20	3	40	50	60

10	40	20	50	3	60

$$a[\underbrace{1 \ldots 2}_{\text{row}},\ \underbrace{1 \ldots 3}_{\text{column}}]$$

1 → lower, 2 → upper, 1 → lower, 3 → upper

no of row = upper − lower + 1 = 2 − 1 + 1 = 2

no of column = upper − lower + 1 = 3 − 1 + 1 = 3

a [2 ... 5 7 ... 12]

2 → l, 5 → u, 7 → l, 12 → u

no of row = 5 − 2 + 1 = 4
no of colum = 12 − 7 + 1 = 6

a [1 ... 2 , 1 ... 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] = $b + [(i - l_r) * n + (J - l_c)] * \omega$

Date: / /
Page No.

Column major

Address of A[i][j] = $b + [(i - l_r) + (j - l_c) * m] * \omega$.

i = row of element to be found
j = column " " " "
b → base address.
ω = size of element.
l_r = lower bound of row
l_c = lower bound of column

Matrix A[4][5] ba = 1020 ω = 2 byte

A[3][4]

i = 3 l_c = 0
j = 4 l_r = 0.
b = 1020 m = 4
ω = 2 byte n = 5

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

$= 1020 + [(3-0) * 5 + (4-0)] 2$.
$= 1020 + [15 + 4] * 2$.
~~= 1020 + 17 * 2~~ = 1058
= ~~1052~~

15
More Examples on 2D

Date: / /
Page No.

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

A[15 20]

$l_r = -15$
$u_r = 10$
$l_c = 15$
u_c 40

$m = 10 + 15 + 1 = 26$ $i = 15$
$n = 40 - 15 + 1 = 26$. $j = 20$
$ba = 1500$
$\omega = 1$ byte

40
15
25

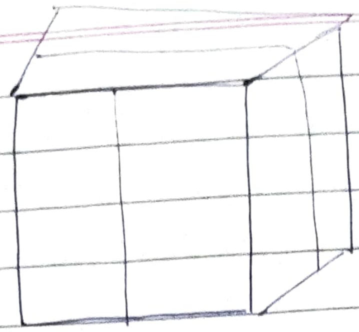
$RM = 1500 + [(15 - (-15)) * 26 + (20 - 15)] * 1$
$= 2285$.

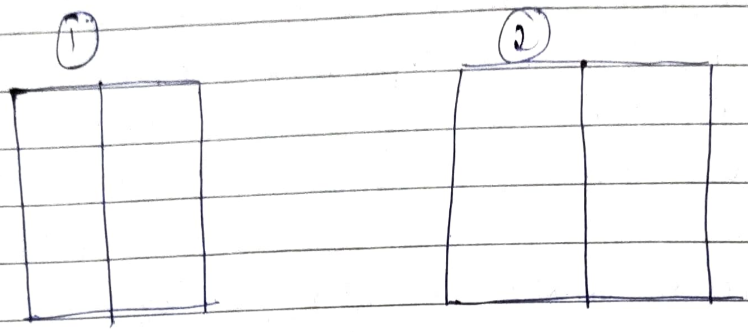
(16)

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

0	1
2	3
4	5
6	7
8	9

②

10	11
12	13
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	19

17

Formula for Row major & column major

Date: / /
Page No.

Consider 3D Array.

A [1 ... P , 1 ... R , 1 ... C].

l_1 u_1 l u_l u

(1) Find length of all dimensions

$L_1 = p - 1 + 1$ upperbound - lowerbound + 1

$L_2 = R - 1 + 1$

$L_3 = C - 1 + 1$

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

$E_i = k_i -$ lower bound

Rowmajor = $Loc\ A[k_1 k_2 k_3] = b + co((E_1 L_2 + E_2) L_3 + E_3)]$

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

Row

$L_1 * L_2 * L_3$

$E_1 \quad + E_2 \quad + E_3$

Column

$L_1 * L_2 * L_3$

$E_1 + \quad E_2 + \quad E_3$

A[1 .. 2, 1 .. 5, 1 .. 2] $\quad k_1 = 1 \quad L_1 = 2 \quad E_1 = 0$

$k_2 = 3 \quad L_2 = 5 \quad E_2 = 2$

b = 100 $\quad$ co = 2 $\quad k_3 = 1 \quad L_3 = 2 \quad E_3 = 0$

RM $\quad A[1, 3, 1] = 100 + 2[(0 \times 5 + 2)\ 2 + 0]$

$= 108$

Date: / /
Page No.

(19)

Multidimensional Array.

A n dimensional array has n subscript

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

The element A with subscript denoted as

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

The programming lang. will store array in

(1) Row major order.

$LOC(A[K_1, K_2 \cdots K_n]) = b + w[((E_1 L_2 + E_2) L_3 + E_3) L_4 + E_{n-1}) L_n + E_n]$.

(2) Column major order

$LOC(A[K_1 K_2 \cdots K_n]) = b + w[((E_n L_{n-1} + E_{n-1}) L_{n-2} + \cdots E_3) L_2 + E_2) L_1 + E_1]$

(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

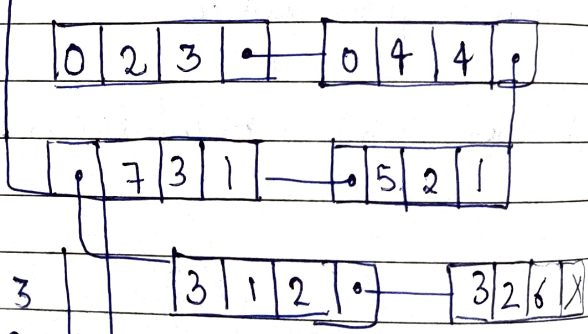
2) computing time

Representation

1) Array represent^n.
Will represent in three field row, column & value

Linked list represent^n
Will represent in 4 field
row, column, value, next node

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & 2 & 6 & 0 \end{bmatrix}$$



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

22

Operations on linear Array

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

1) Traversing

If we want to print/count element of linear Array

LA:

10	20	12	13	15
0	1	2	3	4

LB ↙ 0 ... 4 ↘ UB

$O(n)$

Algo

1) Set K = LB
2) Repeat step 3 and 4 while K<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 opertn 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

10	20	15	12		
1	2	3	4	5	6

n element

	10	20	15	12	
1	2	3	4	5	6

11	10	20	15	12	
1	2	3	4	5	6

n+1 element

At given loc.

10	20	15	12		
1	2	3	4	5	6

11 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 ≥ 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) Del^n from begining O(n) Worst
2) Del^n from end O(1) Best
3) Del^n from given loc^n. avg case O(n)

30	40	25	27	35
1	2	3	4	5

$n = 5$

from beg

40	25	27	35	
1	2	3	4	5

$n-1 = 4$ elements

Worst case : $O(n)$

from end

30	40	25	27	

$O(1)$ best case

n--

from given pos

30	40	~~25~~	27	35

$n-1 = 4$ elements

$K = pos$.

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 refer to the opertn. of rearranging elements of Arrays in increasing order.

A					
	1	2	3	4	5

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

Bubble sort - simplest way to sort an array

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

- 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 (n-1)
- Compare $A[1]$ and $A[2]$ and arrange in order $A[1]<A[2]$
- Compare $A[2]$,, $A[3]$ — $A[2]<A[3]$
- ,,
- ,, $A[N-1]$,, $A[N]$. — $A[N-1]<A[N]$

Pass 2 (n-2)

Pass 3 (n-3)

⋮

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

20	19	13	25	15
1	2	3	4	5

Pass 1
4 comp

19	20	13	25	15

19	13	20	25	15.

19	13	20	15	25

Pass 2

13	19	20	15	25

13	19	15	20	25

n-2 ↳ 3 comparisions.

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

Date: / /
Page No.

(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
	0	1	2	3	4

↑ ↑ ↑

item = 15

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)

```
# 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

1) As a input we need to given sorted i/p array.

Date: / /
Page No.

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

1. BEG = LB   END = UB
2. MID = ((LB + UB)/2)
3. Repeat step 3 and 4 while BEG ≤ END and A[MID] ≠ Item
4. if Item < A[mid] then
   set END = MID-1
   else
   set BEG = MID+1
5. set MID = ((BEG + 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 %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.

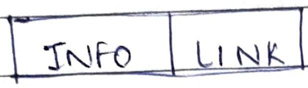
$\log_2 16 = 4$ comparisions gives O/p.

27

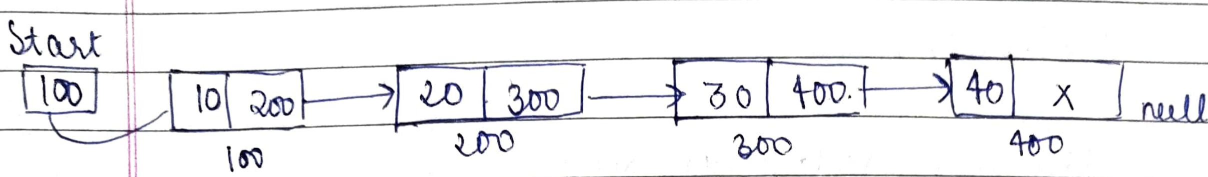
Linked list

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



node



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.

28

Pointer 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 ("\n1. create \n");
 printf ("\n2. display \n");
 printf ("\n3. exit \n");

printf ("Enter your choice")
scanf ("%d", &choice);
switch (choice)
   {
   case1: create();
              break;
   case2: display();
              break;
   case3: exit(0);
              break;
  default; printf("wrong choice");
  }
return 0;
}
```

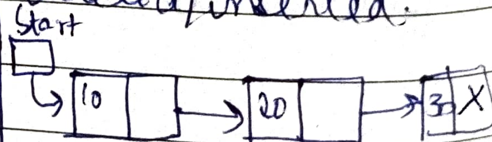
SPS
Date: / /
Page No.

```
void create ()
{
struct node *temp, *ptr;
 temp =(struct node*) malloc (size of (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

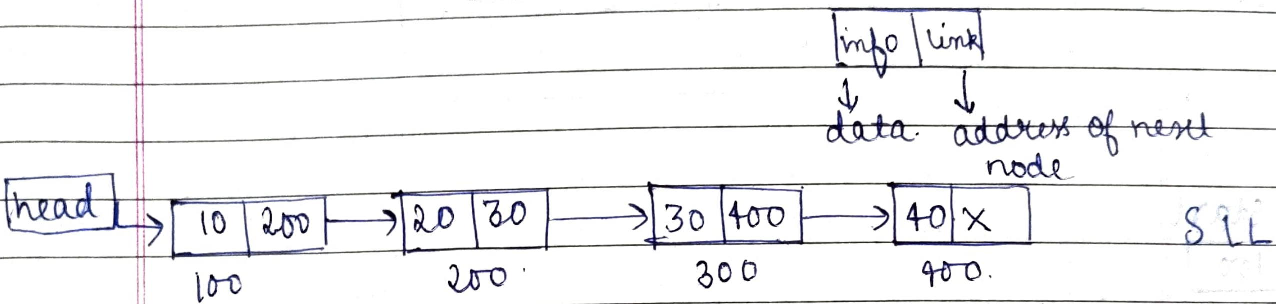
Differences between linked list and Array.

Array	Linked List
1) Array is a collection of similar data.	1) Linked 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^n is easy in LL.
5) Memory allocation during compile time. (Static memory allocation)	5) Memory allocation during run time (dynamic memory allocation)
6) Size of array must be specified at time of declaration.	6) Size of linked list shrink and grow when a new element deleted/inserted.
7) A: | 10 | 20 | 30 | 40 | 50 | 60 | 70 |	

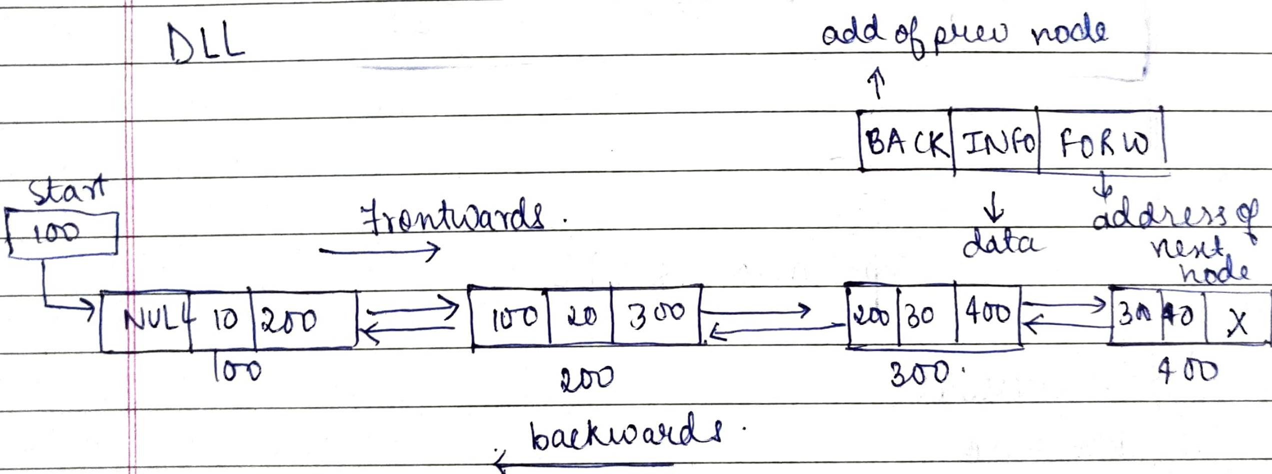
(31)

Types of linked list

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



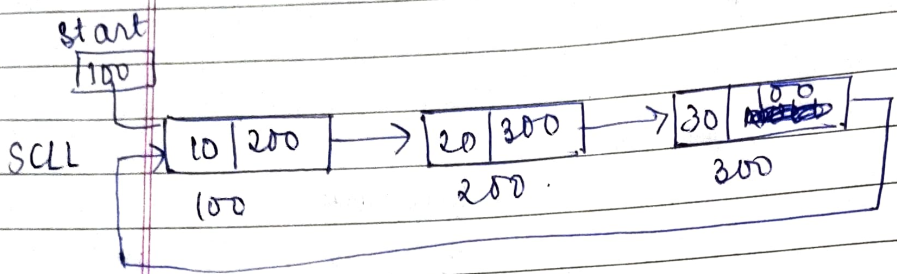
DLL

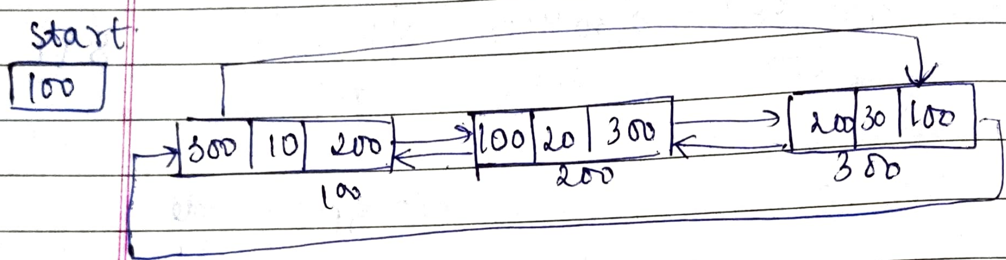



```
Struct node {
 int info;
 struct node * forw;
 struct node * back;
}
```

CLL

Singly CLL   doubly CLL





(32)

Operations on LL.

1) Traversal
2) Insertion
3) Deletion

Traversal

1) Start with head of first & access data
2) Goto the next node and access data
3) Continue untill last node.

SPS
Date: / /
Page No.

Algo

1. Set PTR = START
2. Repeat step 3 and 4 untill 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) {
   printf ( "%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.

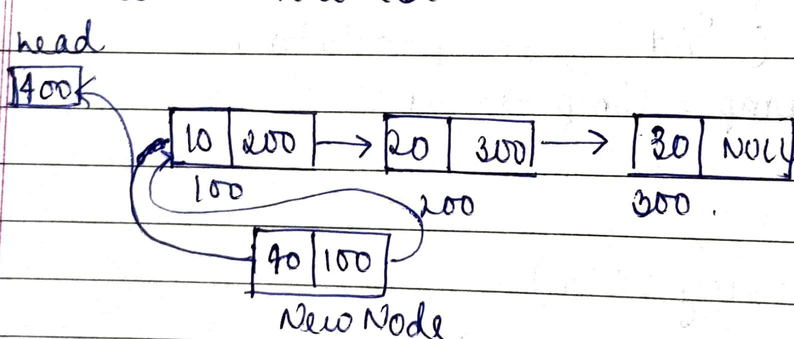
Date: / /
Page No.

Add to begining

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

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

Struct node * newNode;
newNode = malloc( size of ( struct Node));
newNode -> data = 40;
new Node -> next = head;
head = new Node
```

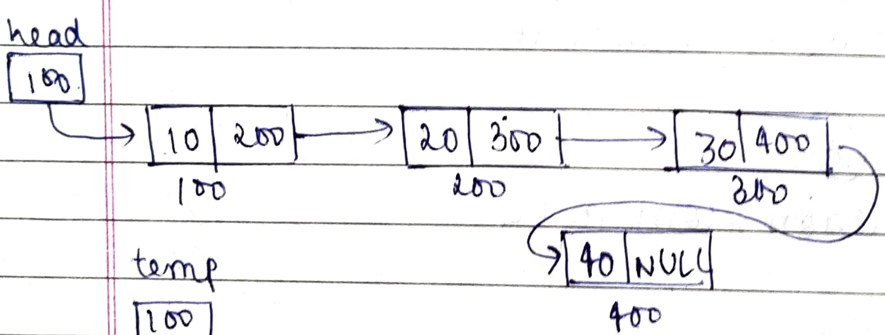


Add at the end.

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

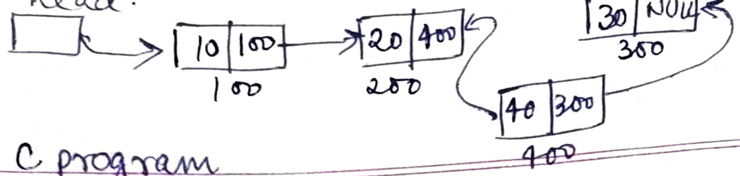
C program

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

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

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



Add^n 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 begining
Point head to second node
head = head -> next

b) From end
1) Traverse to second last element
2) Change its next pointer to null.

```
struct node *temp = head;
while (temp → next → next != null)
        temp = temp → next;
temp → next = null;
```

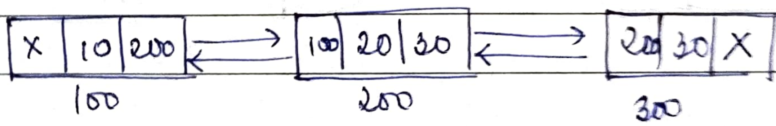
c) From middle or pos.
1) Traverse to element before the element to be deleted
2) Change the next pointer.

```
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
a) Insertion
b) deletion
c) Traversal.

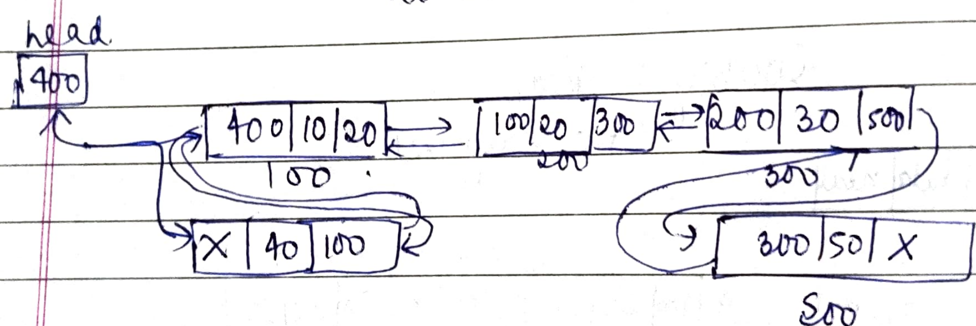
Memory representation.

Start
100

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

1) Traversal
- Forward (next)
- backward (prev)

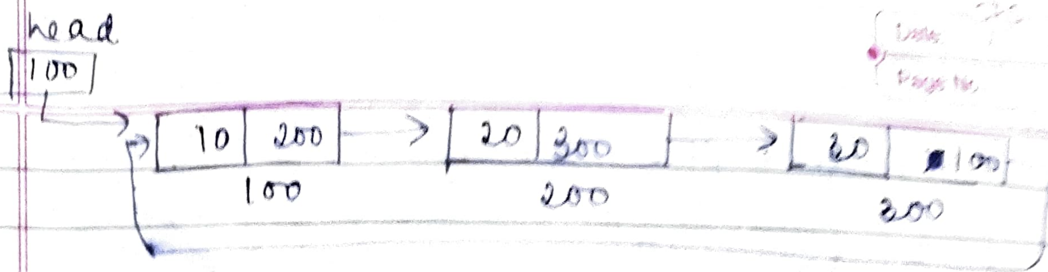
2) Insertion
- beg
- end
- at locⁿ



3) deletion
- beg
- end
- at locⁿ

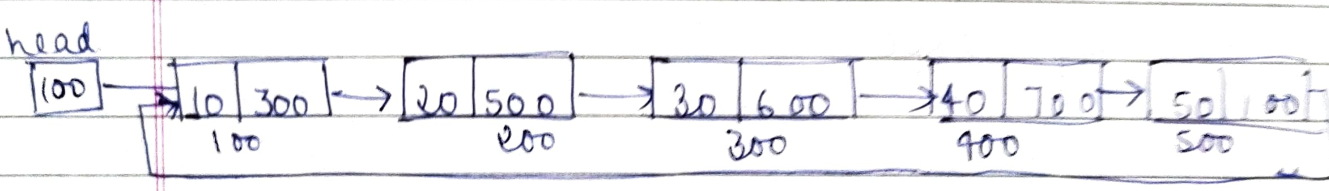
(36)

Circular linked list



Memory representation

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



Operations

1) Traversal

10	20	30	40	50

2) Insertion
a) beg
b) end
c) At pos.

3) Deletion
a) beg
b) end
c) at pos

(37)

Polynomial representation using LL

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

coeff	expo	link

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

head → [2 | 2 | •] → [4 | 1 | •] → [7 | 0 | X]

```
struct node {
 int coeff;
  int expo;
  struct node* next;
  };
```

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

coeff	expo.
5	4
7	2
32	0.

[5 | 4 | •] → [7 | 2 | •] → [32 | 0 | X]

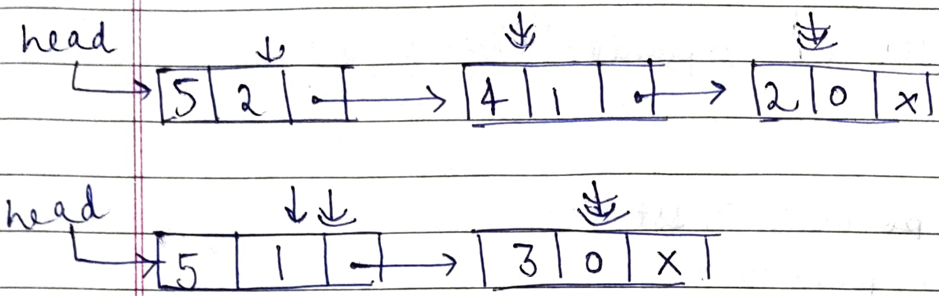
Addition of polynomial

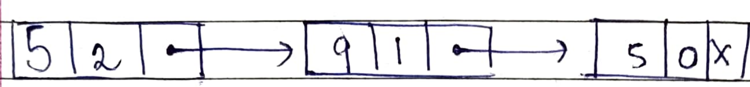
1) Loop around all values of linked list
2) If value of node exponent is greater copy this node to result and head point it.
3) If the value of both exp is same add coeff and add to result.
4) Print result.

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

$p(2) \qquad\quad 5x + 3$

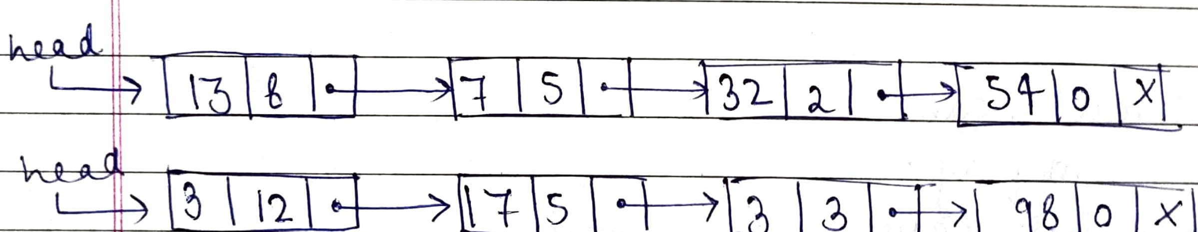
$\qquad 5x^2 + 9x + 5.$



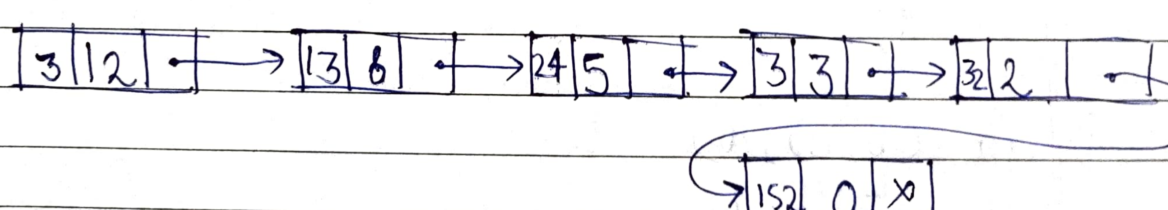


$p(1): 13x^8 + 7x^5 + 32x^2 + 54$

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



98
54
152



38

Generalized LL.

A Generalized linked list is defined as a sequence of $n >= 0$ elements $l_1, l_2, l_3 \cdots l_n$ such as l_i are either atom or list of atoms

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

Flag	Data	Down pointer	Next pointer

Flag : 0 → next pointer exist
1 → down pointer "

Data : atom

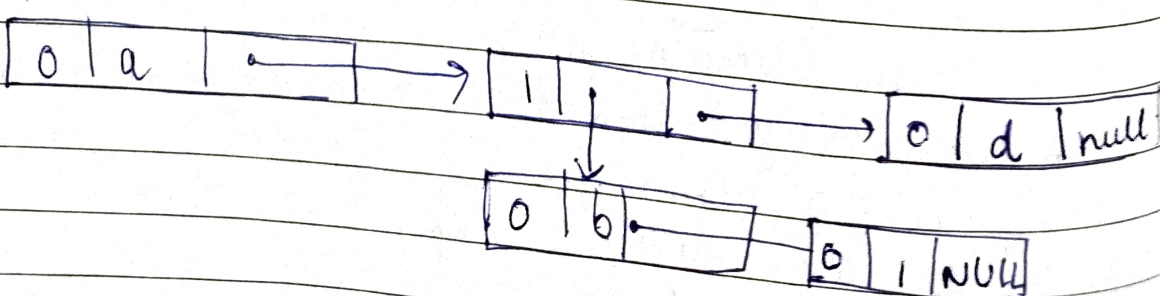
Down p : address to down node.

Next p : address to next node.

```
Struct node
 int flag;
 char data;
 Struct node* down, *next;
 };
```

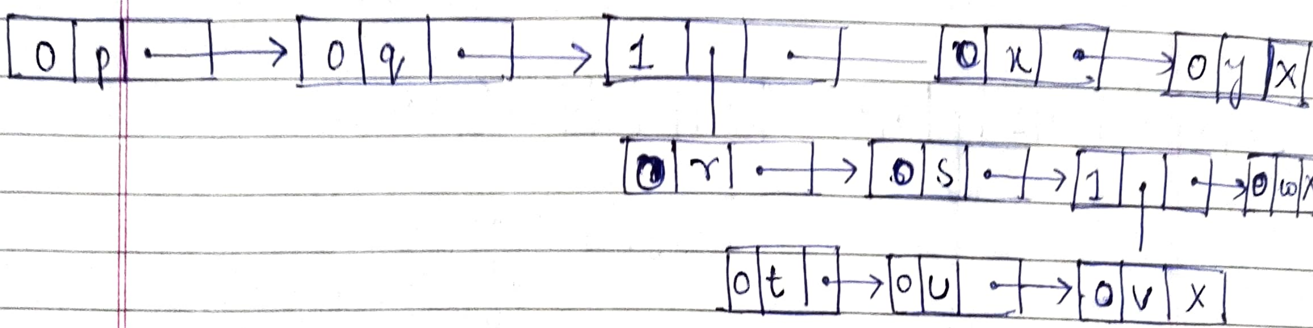
1) (a (b, c), d) = L

① list of atoms ② ③



b) (p, q (r, s (t u v), w) x, y)

① ② ③ ④ ⑤



39

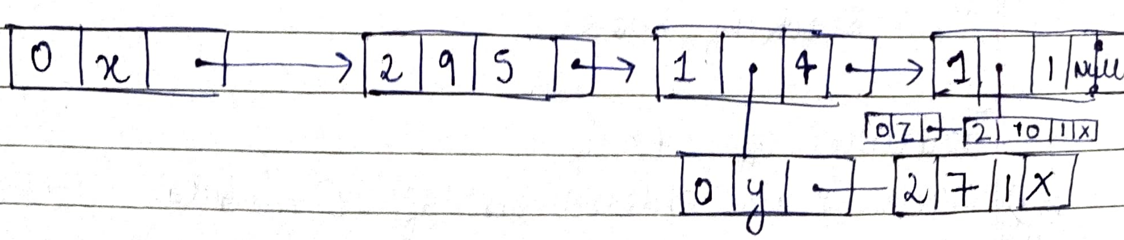
Multivariable Polynomial

We used generalized LL to represent multivariable poly.

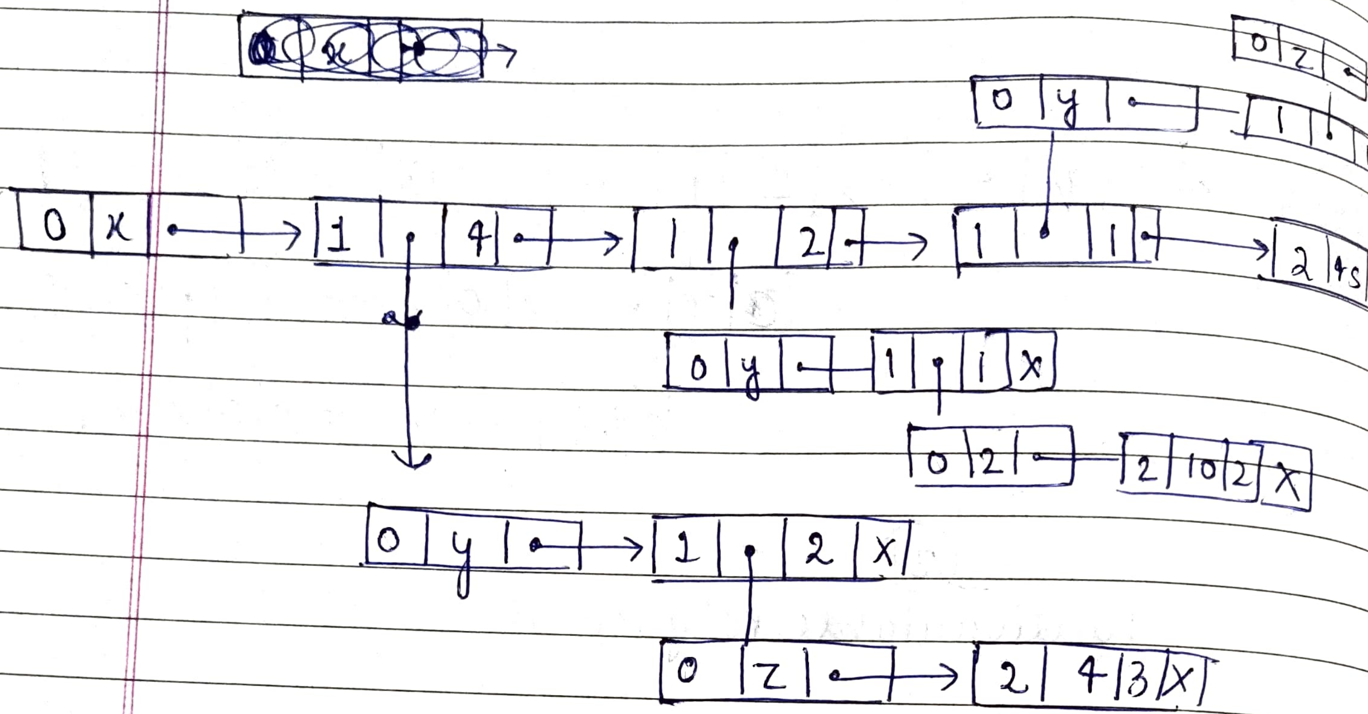
$$9x^5 + 7x^4y + 10xz$$

| Flag | data | down p | next p |

Flag: 0: variable present
1: down variable present
2: coeff exp. present.



Q. $-4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45x^0y^0z^0$.



40

Stack in Datastructure

A stack is a list of element in which an element may be inserted or deleted only at one end called "TOP" of stack. Stack is some time 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
underflow – empty

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

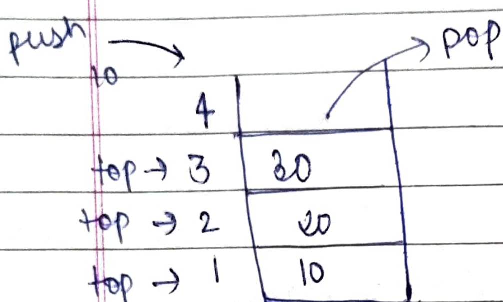
$\frac{11}{2} \times \frac{7}{5}$.

Date / /

Page No.

Applications

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



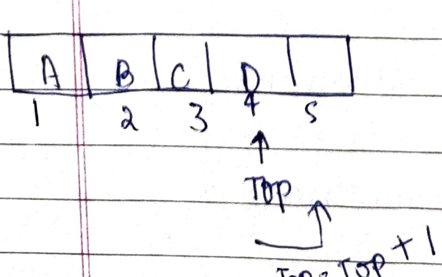
10, 20, 30.

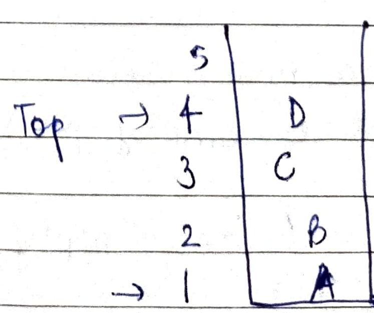
Top = 0

(41)

Array implementation of stack.

Operation of stack
1) Push → insertion
2) Pop → deletion
3) Is empty → is stack empty
4) Is full. → 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)

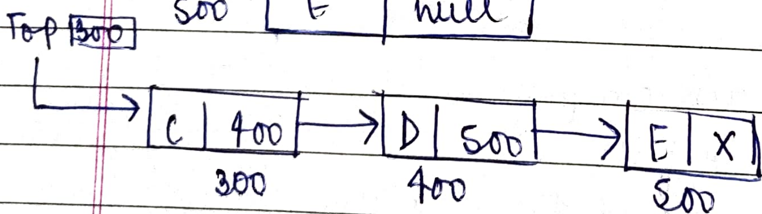
Linked list implementation of stack

Linked list allocate memory dynamically

Top

300	C	400
400	D	500
500	E	null

info	next



Push

1. Create a new node
2. If stack is empty push as start node to list.
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

operator

a + b + c * d

operands.

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

( )

Highest : ↑ (Exponent)
Next highest : * /
Lowest : +, -

( ) > ↑ > (*, /) > (+, -)

2 ↑ 3 + 5 * 2 ↑ 2 − 12 / 6.
8 + 5 * 4 − 12 / 6
8 + 20 − 2.
= 26.

It needs paranthesis & operator precedence

infix : operand1 operator operand 2 (A+B)

postfix : operand1 operand2 operator (AB+)

prefix : operator operand1 operand2 (+AB)

(44)

Infix → 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 '(' onto stack and add ')' to end of Q.
2. scan Q from left to right and Repeat step 3 to 6.
3. If an operand encountered add to P.
4. If left parantheses encounters push to stack.
5. If operator (+) is encountered then
   a) Repeatly pop from stack add to P each operator same or highest precedence.
   b) add (+) stack
6. If Right paranthesis encountered then
   a) Pop from stack & add to P untill left paranthesis
   b) Remove left paranthesis.
7. Exit.

Q. A+(B*C-(D/E↑F)*G)*H

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

45

Examples on Infix to Postfix

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

Symbol	Stack	Postfix Expression
(	(	
A	(	A
*	(*	A
~~B~~	~~(*~~	~~AB~~
~~+~~	~~(+~~	~~AB*~~
~~D~~	~~(+~~	~~AB*D~~
~~)~~		~~AB*D+~~
~~/~~		
(	(*(	A
B	(*(	AB
+	(*(+	AB
D	(*(+	ABD
)	(*	ABD+
/	(/	ABD+*
E	(/	ABD+*E
-	(-	ABD+*E/
F	(-	ABD+*E/F
(	(-(	ABD+*E/F
G	(-(	ABD+*E/FG
+	(-(+	ABD+*E/FG
H	(-(+	ABD+*E/FGH
/	(-(+/	ABD+*E/FGH
K	(-(+/	ABD+*E/FGHK
)	(-	ABD+*E/FGHK/+
)		ABD+*E/FGHK/+-

Date: / /
Page No.

(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)

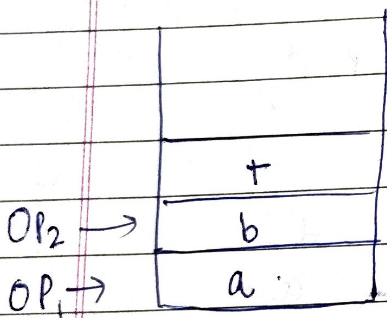
47

Postfix to Infix

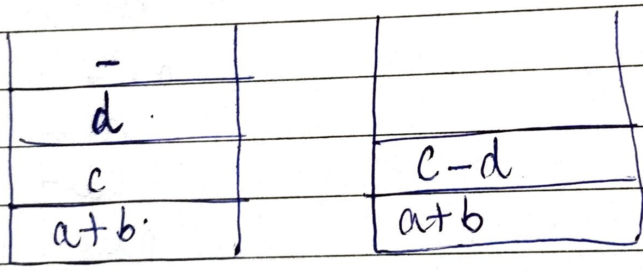
1. Read the postfix expression freom left to right.
2. If we read operand push to stack
3. If we read operator pop Top two
4. go to step 1 untill completed.

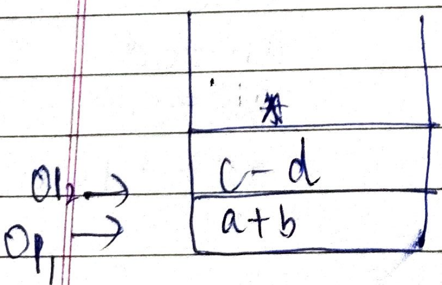
first operand → OP_2
second operand → OP_1
OP_1 operator OP_2

Ex: ab + cd - *



a + b





(a+b) * (c-d)

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 untill complete.

	+
$OP_2 \rightarrow$	b
$OP_1 \rightarrow$	a

+ab.

	−
$OP_2 \rightarrow$	d
$OP_1 \rightarrow$	c
	+ab

−cd.

	*
$OP_2 \rightarrow$	−cd
$OP_1 \rightarrow$	+ab

*+ab−cd.

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 untill complete.

Ex: * + ab - cd

Prefix: *+ ab - cd.

Reverse: dc - ba + *

	-
OP1 →	c
OP2 →	d

c - d

	+
OP1 →	a
OP2 →	b
	c - d

a + b.

	*
OP1 →	a+b
OP2 →	c-d

(c - d) * (a + b)

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-*

oper →	-	cd-
OP1 →	c	
OP2 →	d	

	+	
OP1 →	a	ab+
OP2 →	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 - *

	Stack	
oper →	-	cd-
OP_1 →	c	
OP_2 →	d	

	Stack	
	+	
OP_1 →	a	ab+
OP_2 →	b	
	cd-	

Stack	
*	
ab+	ab+ cd- *
cd -	

49

Evaluation of Postfix Expression

Algorithm.

1) Add ')' at the ~~start~~ end of expression
2) Scan expression from left to Right untill ')' encountered
3) If an operand encountered push to stack
4) If an operator + encountered then
a) POP top 2 operand from stack
First POP operand is OP1
Second POP operand is OP2.
b) evaluate OP 2 + OP1
c) Push to stack.
5) Top of stack is final value.
6) Exit

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

Symbol	Stack	
		OP2 5, OP1 8
5	5	
6	5,6	
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 = 37

(50)

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.

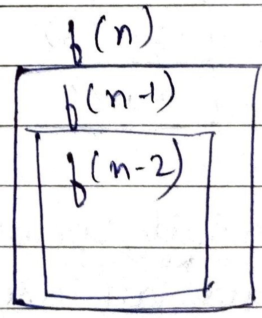
```
fun()
{
 - - -
 - - -
 fun();   -> direct recursion
}
```

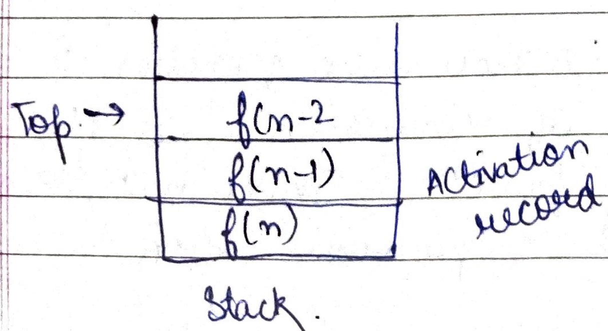
```
fun1()
{ - - -
  - - -
  - - -
  fun2();
}
fun2()
{
  - - -
  - - -
  fun1();
}.
```

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.

```
void fun (int n)                int main()
{                                {
if (n = = 0)                     fun (3);
  return ;                       return 0;
else                             }
printf ("%d", n);
return fun (n-1)
}
```

fun (0) return.
fun (1)
fun (2)
fun (3)
main()

fun 3

tail recursion

3 2 1

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

Date: / /
Page No.

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

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

O/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!.

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

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

$1! = 1$ $\quad 0! = 1$ $\quad 2! = 1 \times 2 = 2$

$3! = 3 \times 2 \times 1 = 6$

$4! = 4 * 3!$

$3! = 3 * 2!$

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

```
int fact (int n)
{
    if (n==0)
        return 1;
    else
    return n* fact(n-1);
}
```

Stack Implementation.

fact(0)	→ return 1
fact (1)	
fact (2)	
fact(3)	
fact (4)	

(53)

Recursion fibonacci series

0, 1, 1, 2, 3, 5, 8, 13 . . . .

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

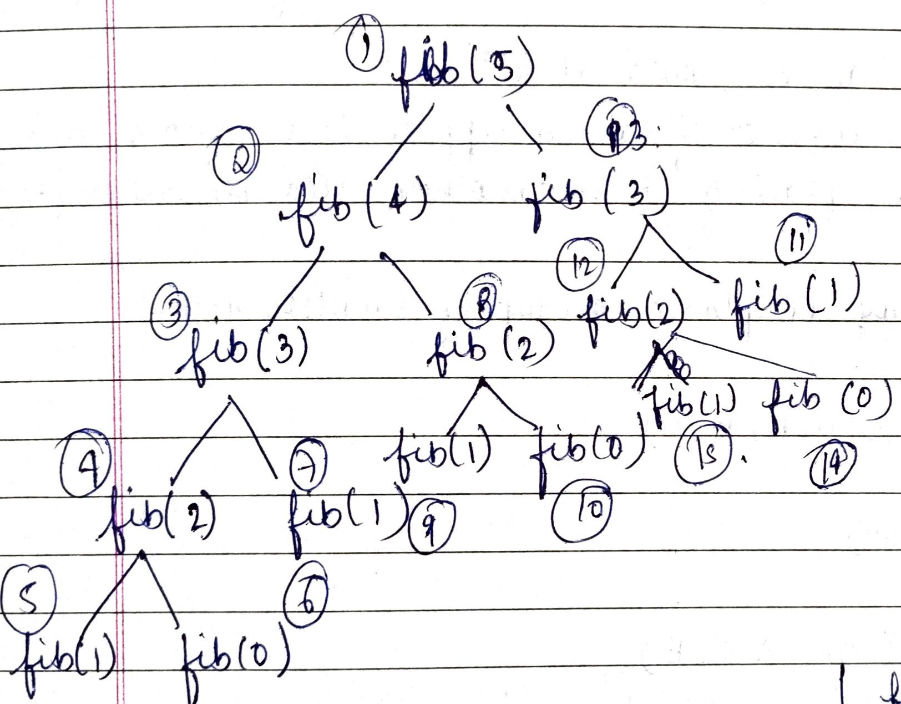
$fib(3) = fib(2) + fib(1) = 1+1 = 2$

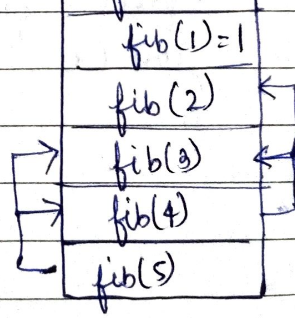
$fib(2) = fib(1) + fib(0) = 1+0 = 1$

$fib(n) = fib(n-1) + fib(n-2)$ → general formula.

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

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





(54)

Tower of Hanoi

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

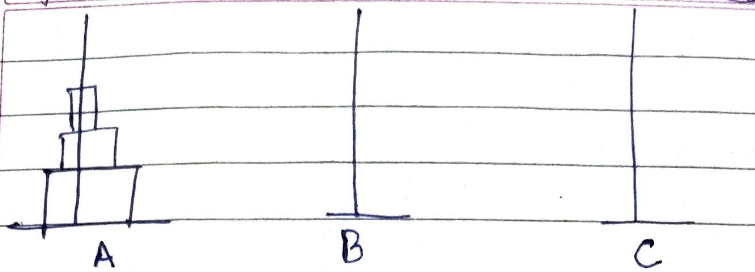
Tower of Hanoi is a mathematical puzzle invented by Mathematician Lucas in 1883. In this puzzle we have 3 ~~[illegible]~~ rods and n disk Objective puzzle to move entire disk from first rod to another by following

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)

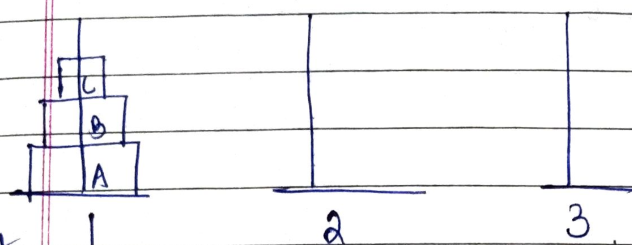
A → from, B → to, C → using



Taking an example of 3 disk

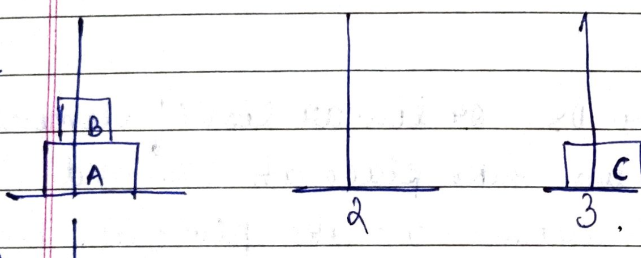
next

1→3



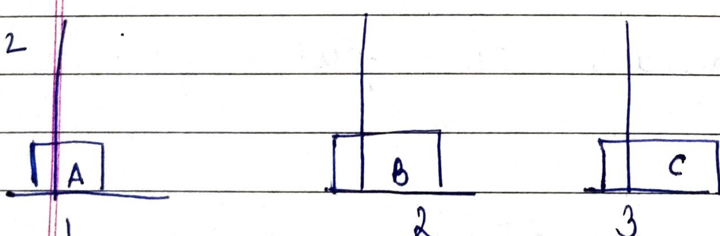
next

1→2



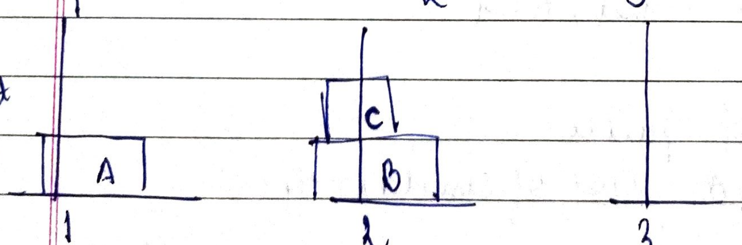
next

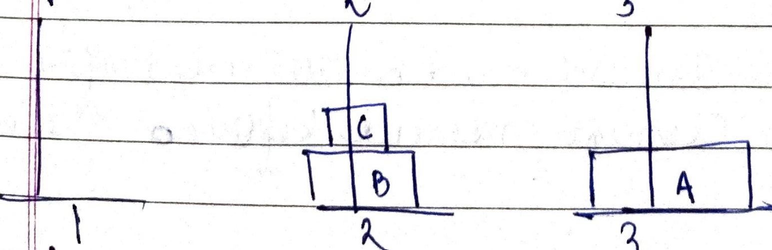
3→2

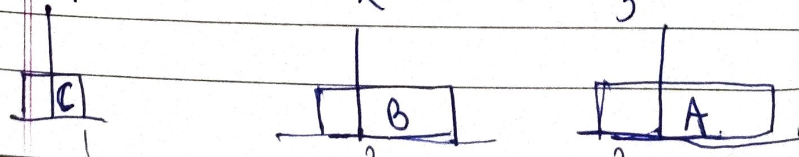


next

1→3



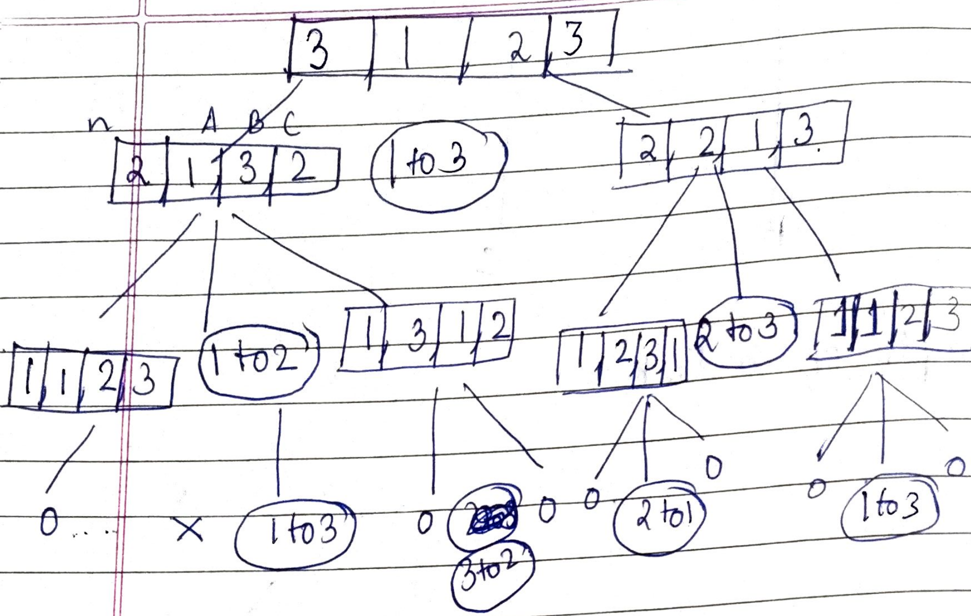




$$\left.\begin{matrix}(1,3)\\(1,2)\\(3,2)\\(1,3)\\(2,1)\\(2,3)\\(1,3)\end{matrix}\right\} 2^n-1$$

no of moves $= 2^n - 1$

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



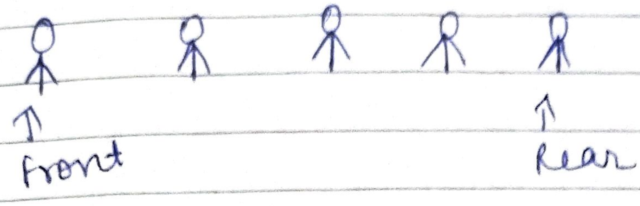
55

Queues

A Queue is a linear DS or linear list of element 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
4) 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

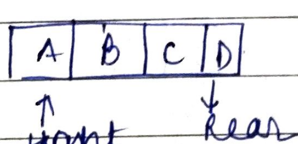
Operations.

1) enqueue() → insertion (Rear)

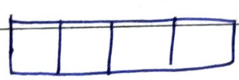
2) dequeue() → deletion (front)

3) Peak() → value of front (peak value)

4) isfull() →



5) is Empty() →

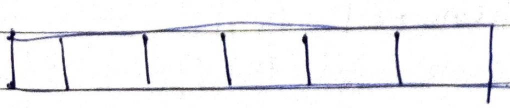


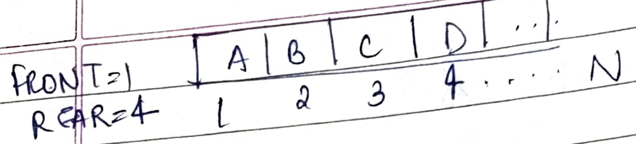
(56)

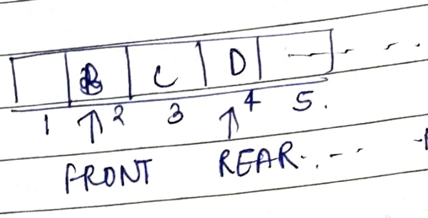
Array representation of Queue.

Queue will maintain by a linear array with the help of 2 pointer "front" & "Rear"

FRONT=REAR=0
NULL



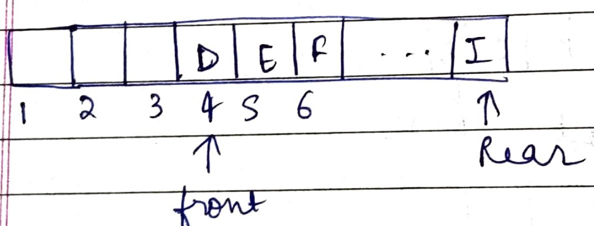




1) FRONT = NULL is Empty → Queue is (underflow)

2) FRONT = FRONT+1 (after ~~insertion~~ deletion)
3) REAR = REAR+1 (after insertion)

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



queue is not full as front ≠ 1

We do indexing of rear.

QINSERT (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[R] = item
6. Return

SPS

Date: / /

Page No.

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

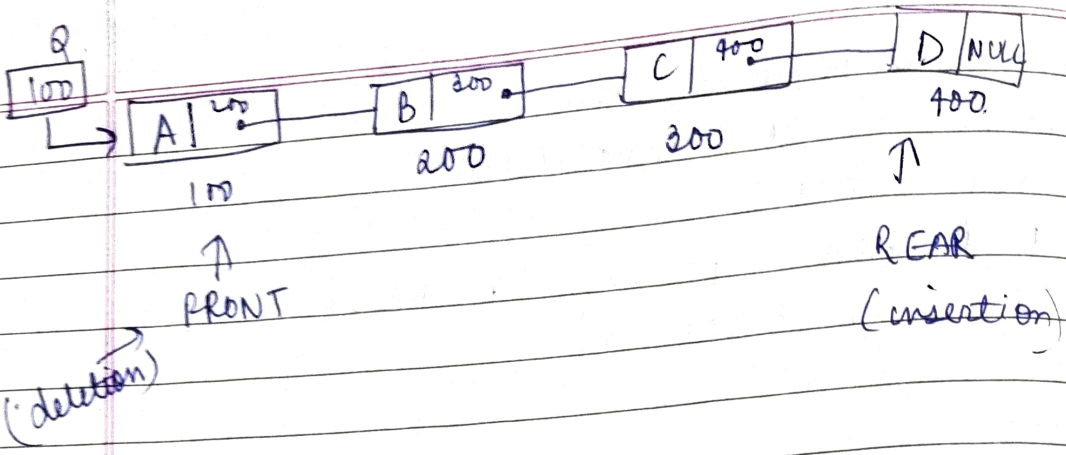
1. if F=NULL write "underflow"
2. Set item = Q[F]
3. if F=R then F=NULL R=NULL
4. else if F=N then set F=1
5. else F=F+1
6. Return

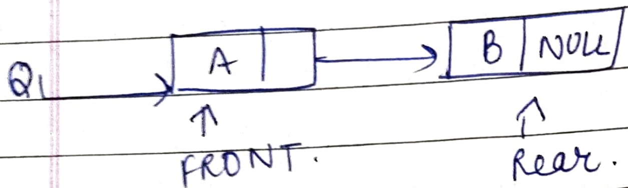
57

Linked list representation of LL.

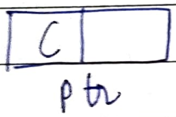
A linked list is implemented using 2 pointers front & Rear. Each node of linked list contain 2 part info & link

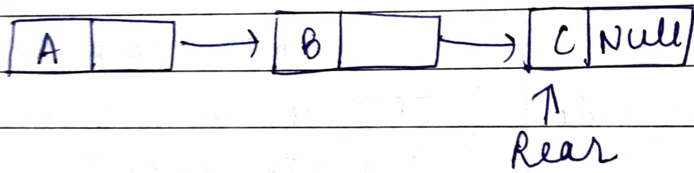
INSERT ALGO	DELETE ALGO
1) Allocate space for new node Ptr.	1) IF FRONT = NULL write underflow
2) Set ptr → info = item	2) Set PTR = FRONT
3) if front = NULL	3) Set FRONT = FRONT → LINK
set FRONT = REAR = Ptr	4) FREE PTR.
Set FRONT → LINK = REAR → LINK = NULL	5) end.
4) else	
set REAR → LINK = PTR.	
Set REAR = PTR.	
set REAR → LINK = NULL	
5) end	

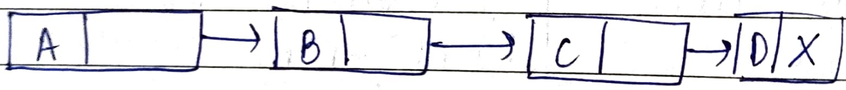




Insert C & D







58

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 FIFO 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 R = -1 set F = R = 0.
else set R = (R+1) % N.
3) set Q[R] = item
4) exit

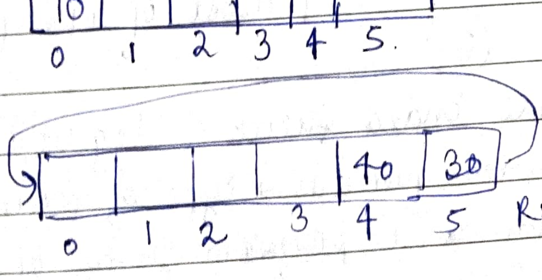
Delete

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

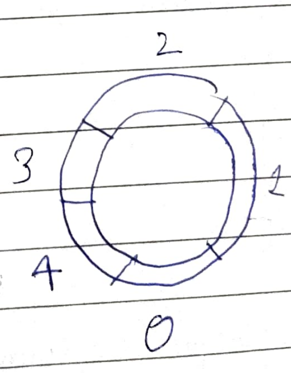
F = R = -1

F = 0
R = 0

F = 3
R = 4



R = n-1



F = -1
R = -1

F = 3
R = 0

(59)

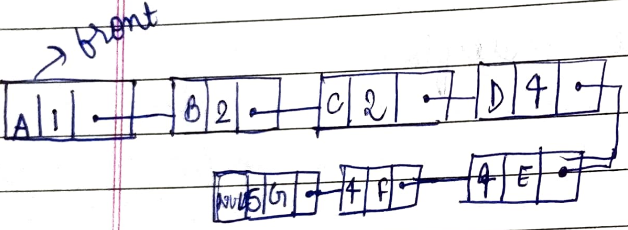
Priority Queue (higher no, lower priority)

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

Representation

1) linked list (one-way)

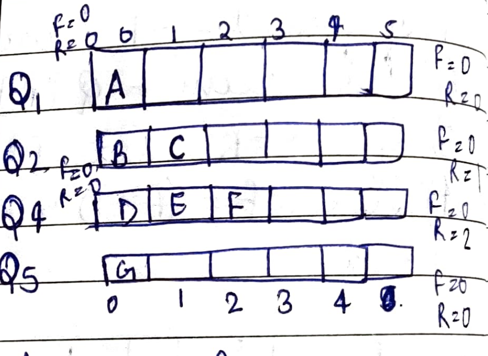
A B C D E F G
1 2 2 4 4 4 5



Deletion will take place from highest priority.

2) Array (multiple

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

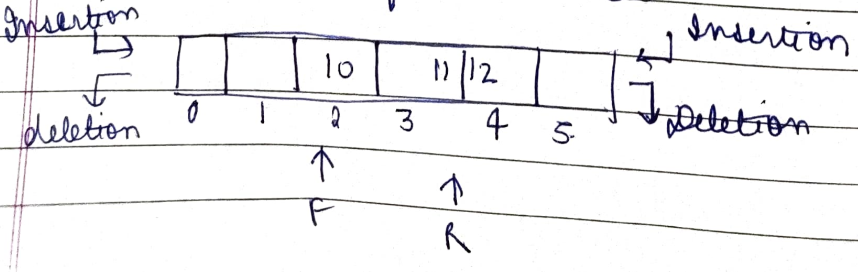


Insertion & deletion using simple queue. deletion starts from queue having most of the priority.

(60)

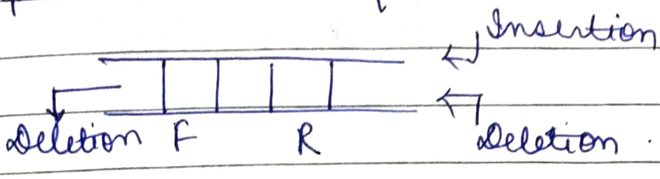
Double Ended Queue (deque)

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



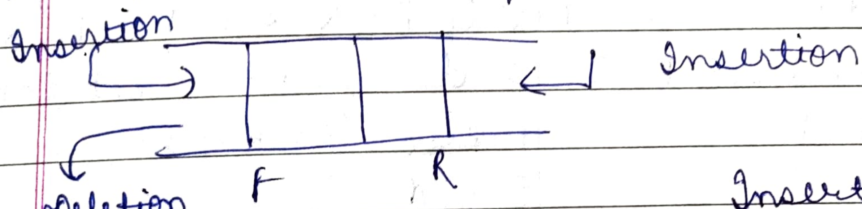
1) Insert at FRONT
2) Insert at REAR
3) Delete from FRONT
4) Delete from REAR

a) Input restricted Deque



Insertion → Rear side
Deletion → Front as well as Rear.

b) Output-restricted queue.



Insertion → Front as well as Rear
Deletion → ~~Rear~~ Front side

Insertion at front takes a circular manner.

10	24	21	13	14	12

↑F (under 13)

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] = item

Insert Rear

1) If F = 0 && R = N−1 or F=R
   write "overflow"
2) If F = −1 set F = R = 0.
3) else if R = N−1 set R = 0
4) else R = R + 1
5) Q [F] = item

Delete FRONT

1. if F = −1 write "underflow"
2. If F = R set F = −1 R = −1
3. else if F = N−1 set F = 0
4. else F = F+1
5. end

Delete REAR

1. If F = −1 write "underflow"
2. If F = R set F = −1, R = −1
3. else if R = 0 set R = N−1
4. else R = R−1
5. end.

F = −1
R = −1

0	1	2	3	4	5

size = 6.

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

F = R = 0

0	1	2	3	4	5
10					

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

F = 5
R = 0

0	1	2	3	4	5
10					12

insert 14 at front

F=4
R=0

0	1	2	3	4	5
10				14	12

(F = F−1)

insert 21 at rear.

F=4
R=2

0	1	2	3	4	5
10	21			14	12

(R = R+1)

delete from front 14

F=5
R=2

0	1	2	3	4	5
10	21			~~14~~	12

↑ F

(F = F+1)

delete from front 12

F=0
R=2

0	1	2	3	4	5
10	21				~~12~~

F = N−1
set F = 0.

delete from rear 21

F=0
R=1

0	1	2	3	4	5
10					

R = R−1

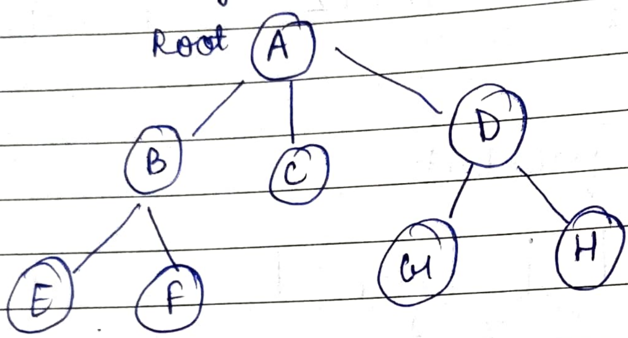
61

Trees.

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

Properties

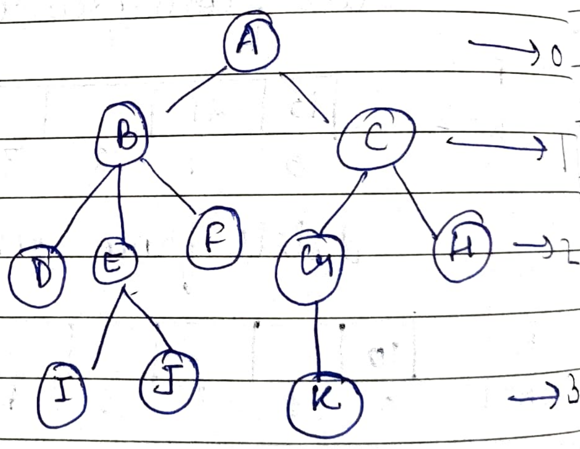
1) There are only one root no parent
2) Except root each node have exactly one parent
3) A node may have zero or more children
4) There are unique path from root to each node.
5) There is no cycle created in tree.



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

Edges: (A-B), (A-C), (A-D), (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

SPS
Date: / /
Page No.

5. Subtree : Subtree (division of tree)

6. degree of node : how many child nodes
d

A → 2   C → 2
B → 3   G → 1
D → 0   H → 0
E → 2   K → 0
F → 0 .   I → 0

7. degree of tree : heighest 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

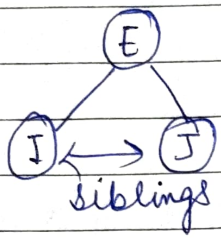
Height of A = leaf node se pahuchne ka longest path. (3)
Height of B = 2.
D = 0
C = 2
K = 1

Depth of B = root se kitna edge hai B pe 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 → J, G → H

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

ancestor for D (ancestors) : 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 trees

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

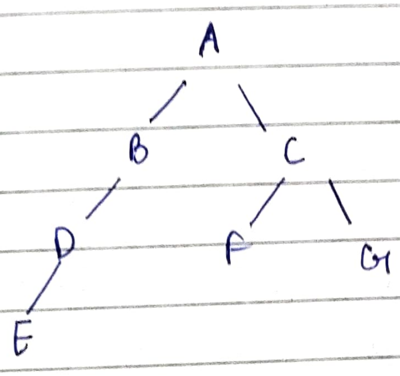
(62)

Binary Tree.

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

Types of Binary tree

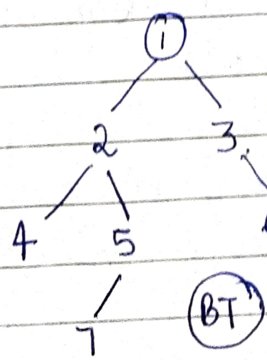
1. Strictly BT (full / proper)
2. Complete BT
3. Perfect BT
4. Balanced BT
5. Extended BT



BT

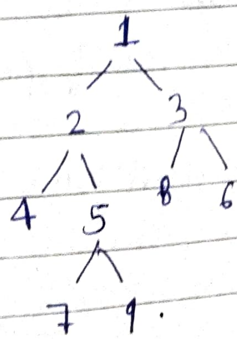
A = 2   D = 1   G = 0.
B = 1   E = 0
C = 2   F = 0.

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



X

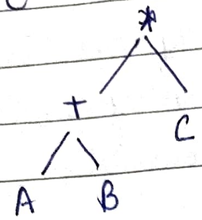
BT



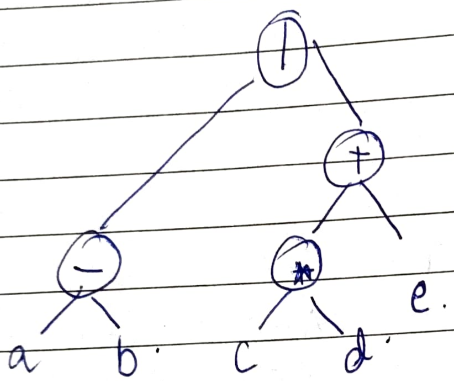
S.B.T

In case of arithmetic calc^n.

(A+B) * C



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



S.B.T

operators → internal nodes
variables → external nodes

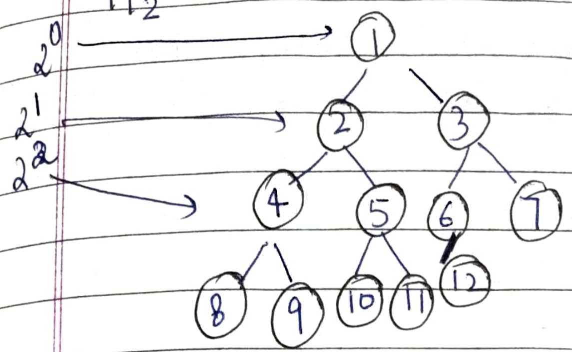
63

Complete BT

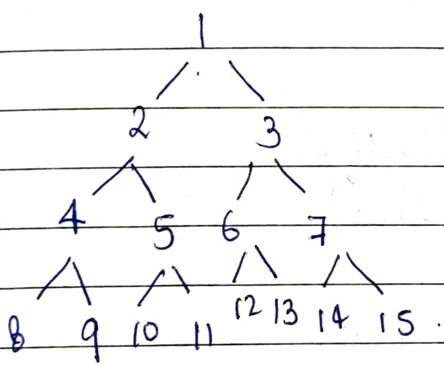
2. Complete BT : The BT is said to be complete if all ~~its~~ level except possibly the last have max^m. no of possible node.

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 $P_n = \lfloor \log_2 n + 1 \rfloor$

TL2



Perfect BT → A complete BT having last level completely filled.

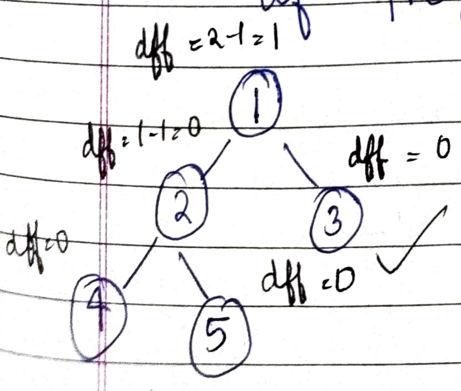


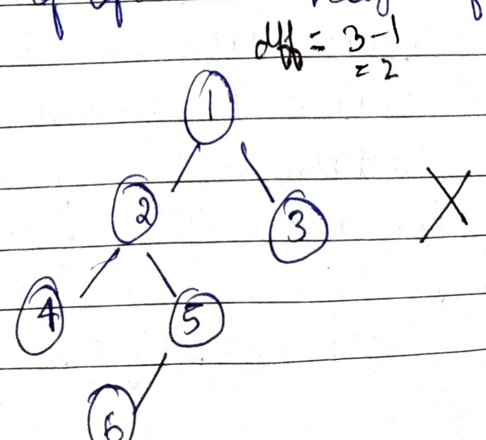
64

Balanced & Extended BT

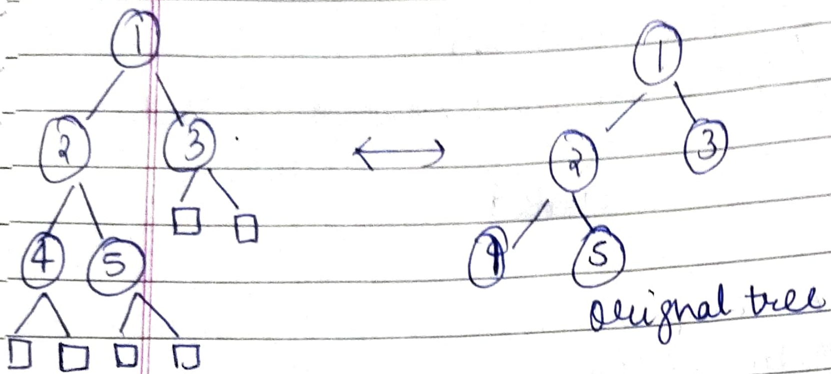
Balanced BT → It is defined as BT in which the height of left & right subtree of any node differ by not more than 1.

df = |height of left child − height of right child|





(2-tree) Extended BT - Extended BT is a type of BT in which all null sub tree at orignal tree are replaced with special node called external node



65

Representation of BT
(Linked list)

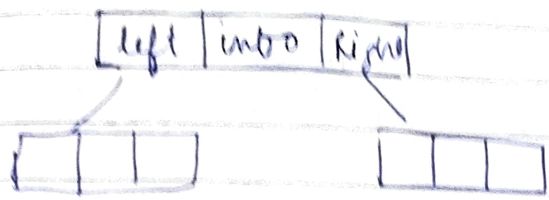
Representation of Binary tree
1) Link representation (linked list)
2) Sequential representation (Array)

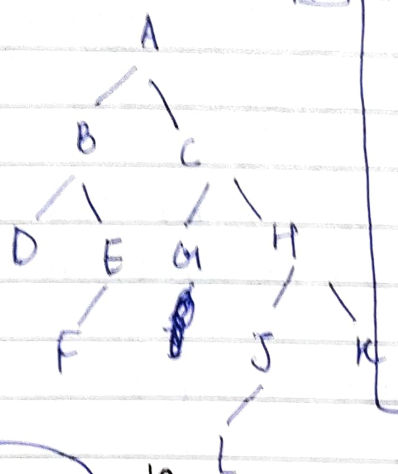
Using ll representation

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.

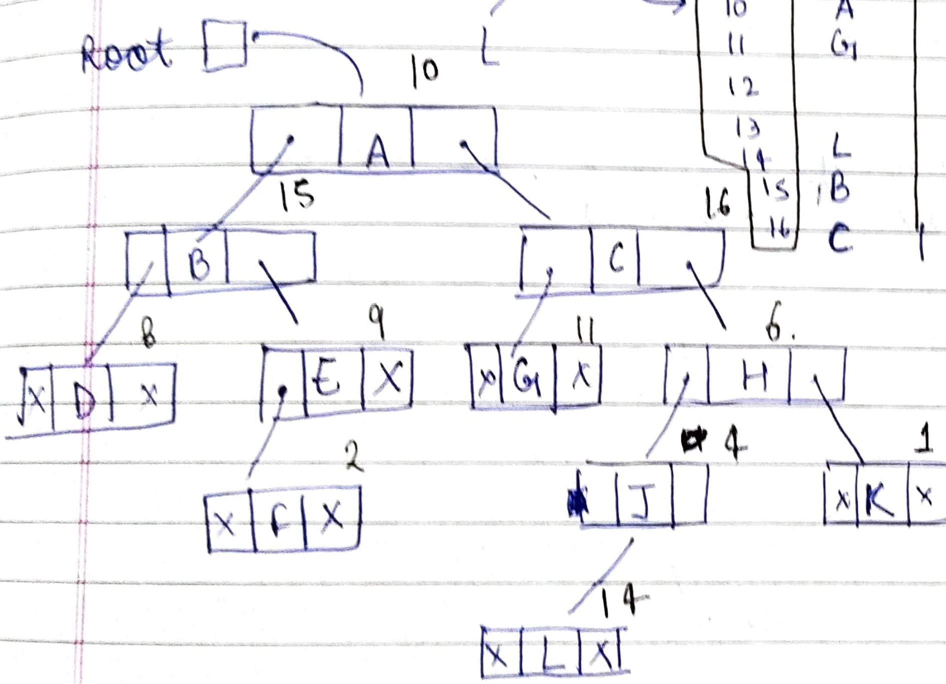
Left	info	Right





Root
10

	Info	left	Right
1	K	0	0
2	F	0	0
3			
4	J	14	0
5			
6	H	4	1
7			
8	D	0	0
9	E	2	0
10	A	15	16
11	G	0	0
12			
13			
14	L	0	0
15	B	8	9
16	C	11	6



66

Sequential Representation (Array)

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

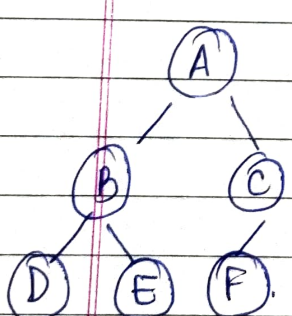
This representation uses simple linear array as:

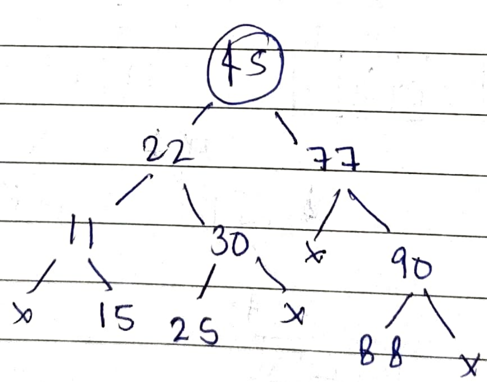
1. The Root of tree stored at T[1]

2. If node N occupies T[K] then its left child is stored at 2K & right child at 2K+1.

3. If T[1] = 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

45	22	77	11	30		90		15	25					88						...	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19		32

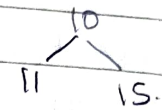
memory wastage

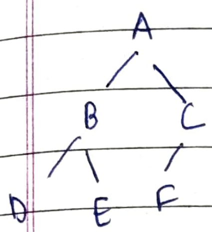
T[k]

1	10
2	11
3	15
4	
5	

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

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





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

A	B	C	D	E	F											
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

B

$2 \times 2 = 4$ (LC)

$2 \times 2 + 1 = 5$ (RC)

C

$2 \times 3 = 6$. (LC)

67

Traversing BT

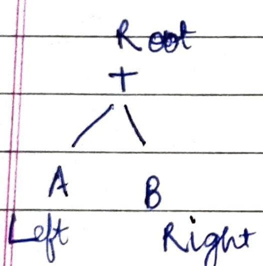
Tree

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

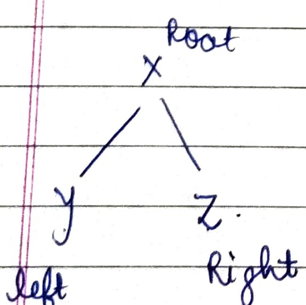
Pre-order
1) Process the Root R.
2) Traverse the left subtree of R in pre order
3) Traverse in the right subtree of R in pre order

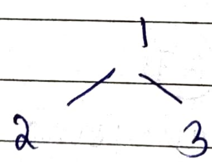
Post order.
1) Traverse the left subtree
2) Traverse the right subtree
3) Process the Root R.

Inorder
1) Traverse the left subtree of R
2) Process the Root
3) Traverse the right sub tree of R in inorder.



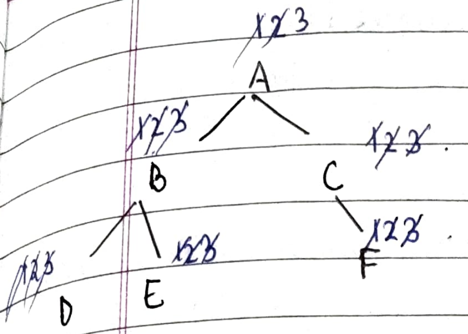
		123
+ A B.	prefix / preorder	X Y Z.
A + B.	infix / inorder	Y X Z
A B +	postfix / postorder	Y Z X



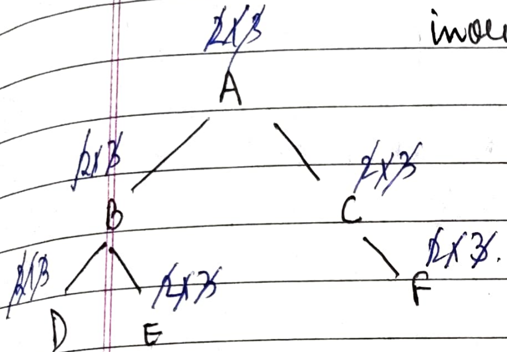


1 2 3
2 1 3.
2 3 1

Now lets find preorder, post order and inorder.

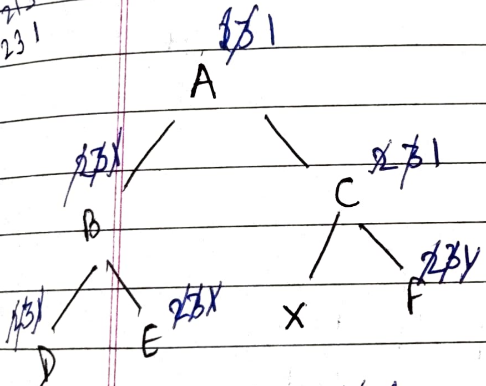


preorder: A B D E C F

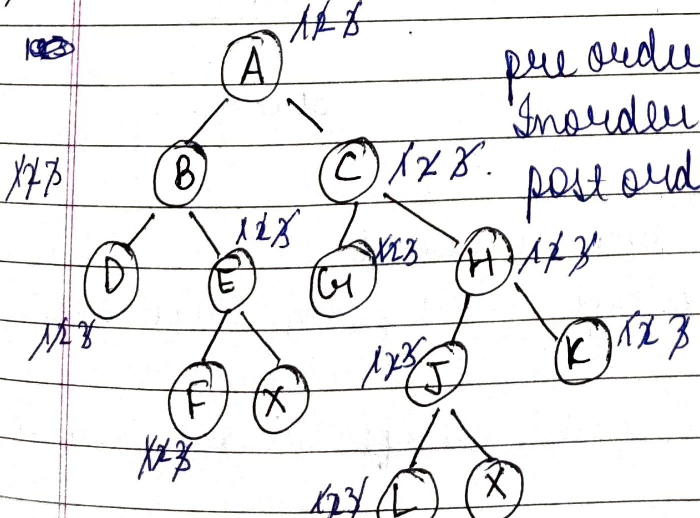


inorder: D B E A C F

123
213
231



postorder: D E B F C A



preorder (123): ABDEFCGHJLK
Inorder (213): DBFEAGCLJHK
postorder (231): DFEBGLJKHCA

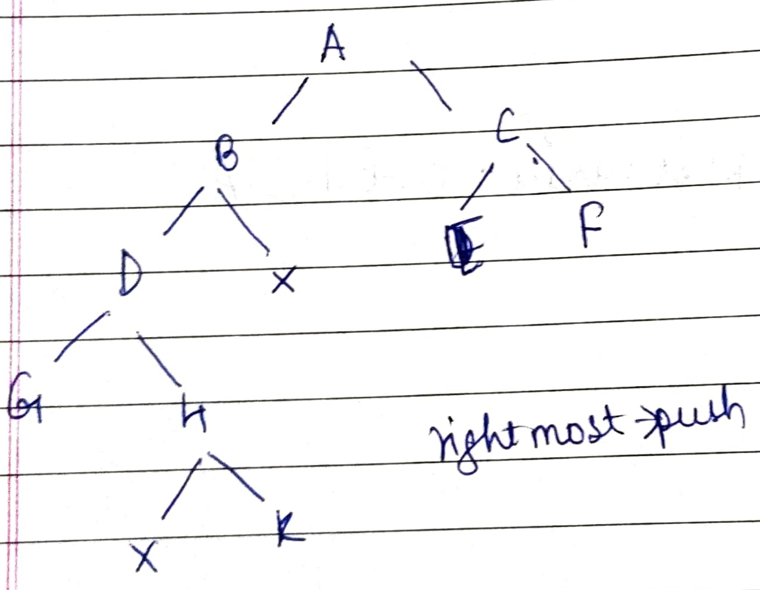
Preorder Tree Traversal

Traversing of B T using stack

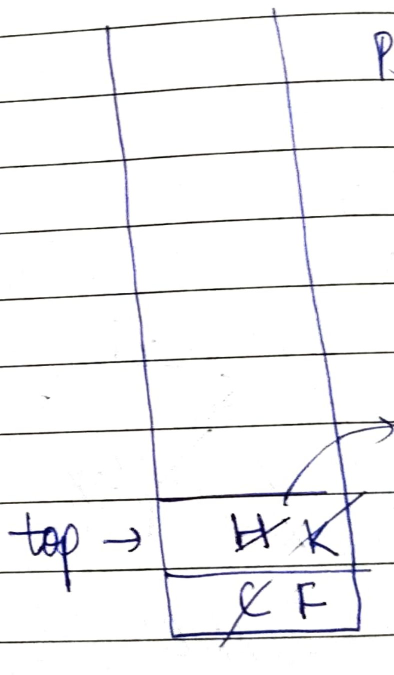
Pre order.

1) Proceed down to left most path, processing each node N on path and push each right child onto the stack. The traversing end after a node N with no left child processed

2) Pop top element on stack, then return to step a if stack is empty exit.



right most → push



Stack.

Preorder:
A, B, D, G,
H, K, C,
F

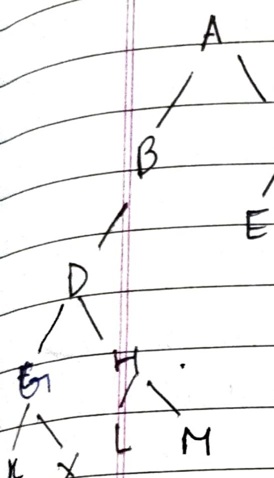
(69)

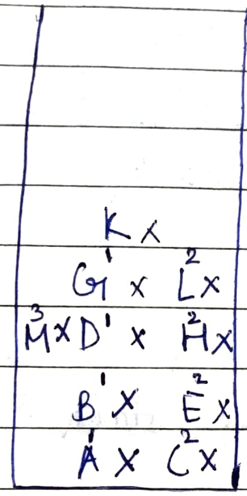
Inorder tree traversal.

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

Stop when node n with no left child pushed on the Stack.

b) Pop & process the node on stack.
1) if null is poped exit
2) if a node N with Right child is processed & Return to step (a).





leftmost → push then pop till right child

Inorder: KGDL, HM, B, A E

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

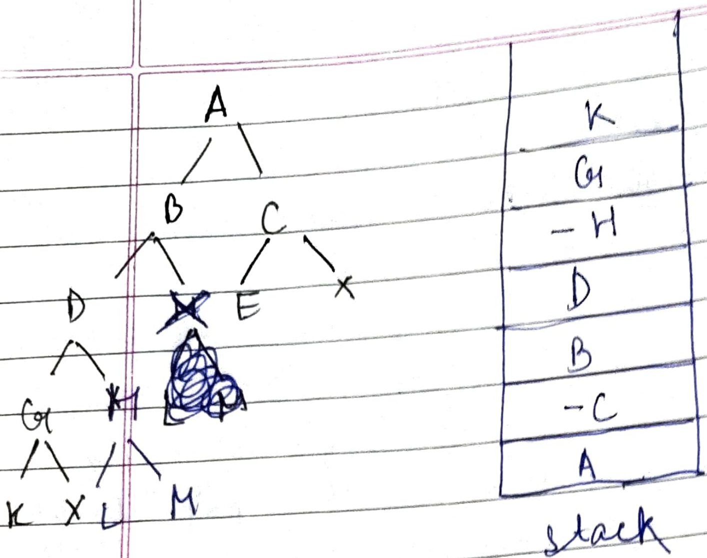
(69)

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.
i) if negative node is popped return to step a.
ii) if null is popped, exit.

Page No.



KGLMHDBECA

left child insert karte jao agar kisi parent ka me -right child ka right child hai to use stack sath push karo.

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

(71)

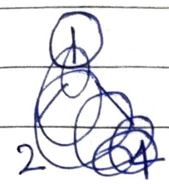
Construction of Binary tree from given traversal.

i) Binary tree from pre & inorder
ii) " " " post & inorder
iii) " " " pre and post

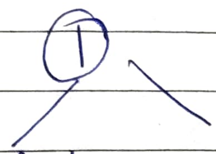
Steps
1) Identify root from preorder.
2) identify element left and right subtree from inorder.
3) Repeat step 1 & 2.

preorder : 1 2 4 3 5 7 8 6
inorder : 4 2 1 7 5 8 3 6

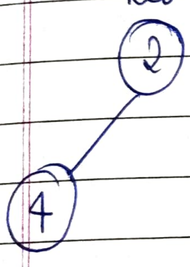
left subtree   right subtree

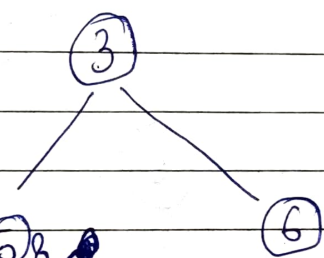


123
213
~~321~~
231

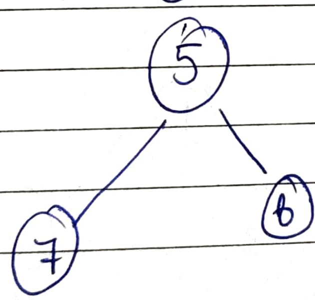


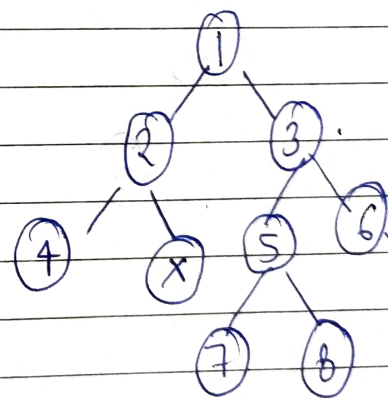
in: 4, 2 Root   in: 7 5 8 3 6.
pre: 2, 4   pre: 3 5 7 8 6.
Root





in: 7, 5, 8
pre: 5 7 8





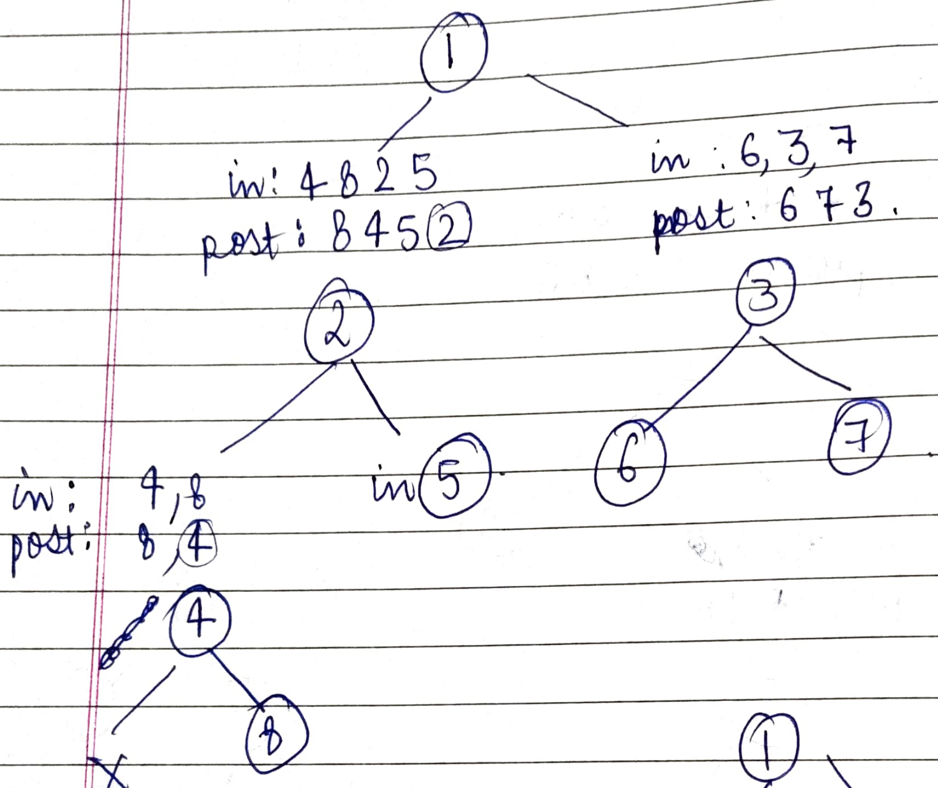
72

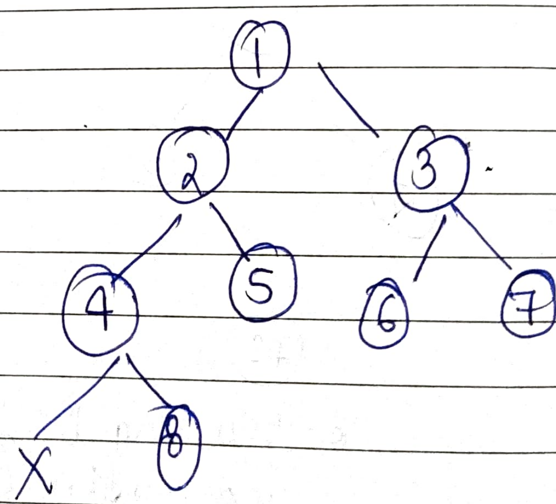
Constructing BT from postorder & Inorder.

Steps :
1) Identify root from post order
2) Identify element 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 : 2 3 1
In : ~~[illegible]~~.
213





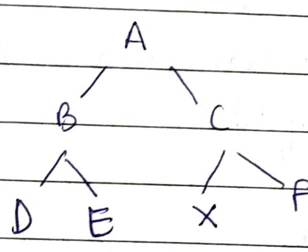
73

Constructing from pre & post order.

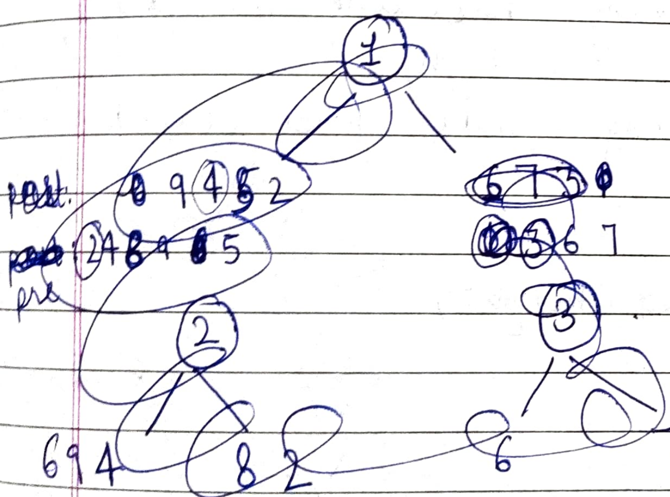
Steps
1) Identify root from pre order
2) Identify left child from pre order
3) Identify left subtree & right subtree from post order
4) Recursively repeat steps for each subtree.

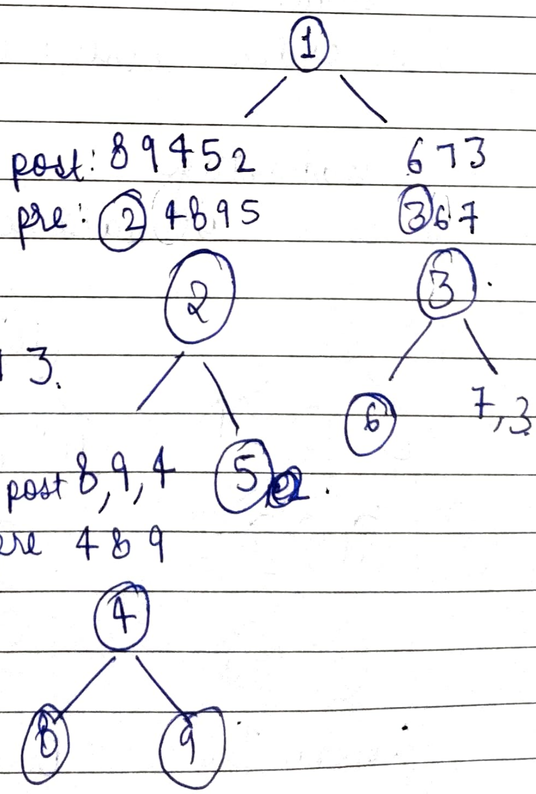
always full BT { Pre order: 1, 2, 4, 8, 9, 5, 3, 6, 7
Post order: 8, 9, 4, 5, 2, 6, 7, 3 1

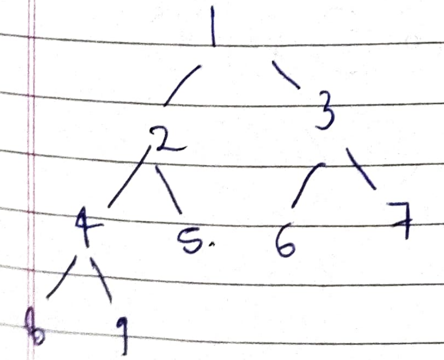
Pre 123
Post 231



Root 2nd
Preorder: AB DECF
Post order: DEB FCA
Left Right







(74)

Binary Search Tree.

Binary search ~~tree~~ tree every node is organi in specific order. This is also called Ordered binary tree.

Suppose T is Binary tree the T is called BST if each node of T has following properties.

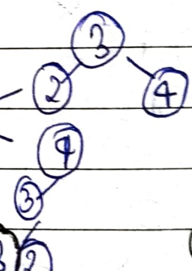
1. The value at N is greater than every value in left subtree.
2. And N is less than every value in Right subtree of N.

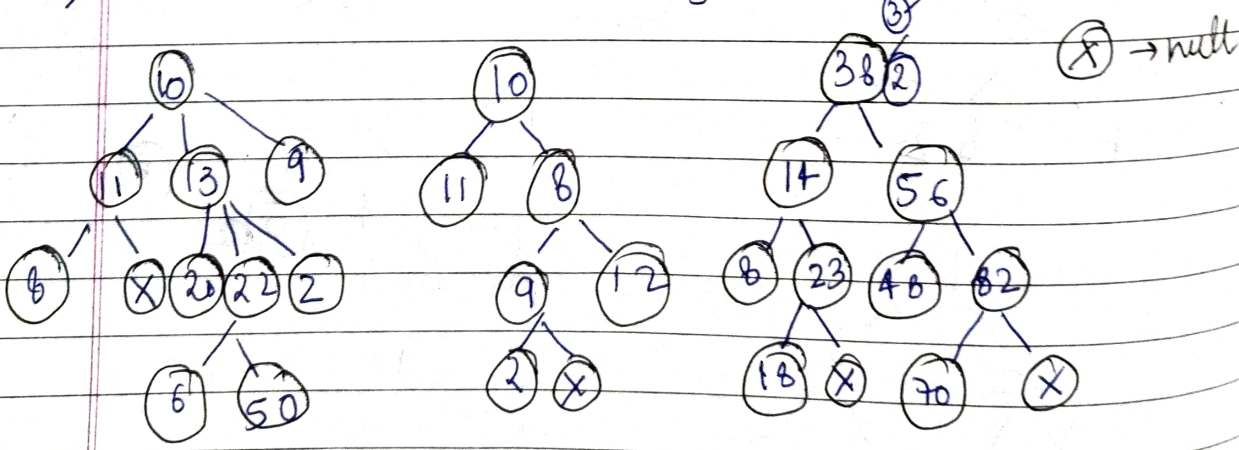
Advantage

1) Avg time of searching, inserting, deleting of element in BST is O(logn)
2) In order traversing always gives sorted array

Disadvantage

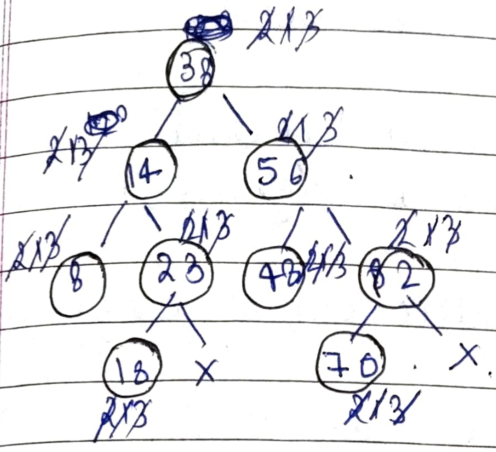
1) It is not balanced (height)





General Tree Binary tree BST

Inorder

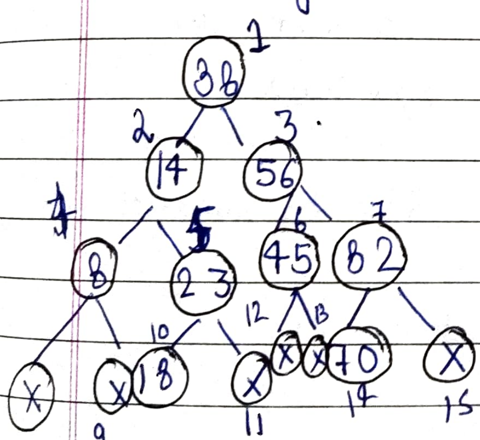


~~8~~, ~~14~~ 18, 38, 48, 56, 70, 82

75

Binary Search tree
Insertion.

Operations performed on BST
1) Insertion
2) Deletion
3) Searching



1	38
2	14
3	56
4	6
5	23
6	45
7	82
8	
9	
10	18.
11	
12	
13	
14	70

2k = left
2k+1 = Right

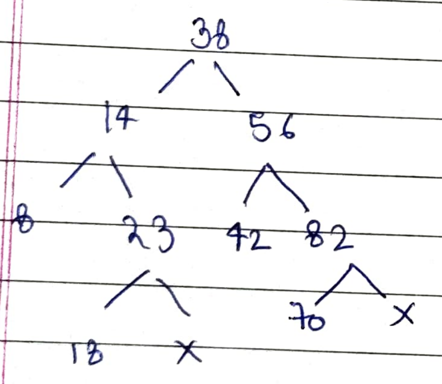
Searching

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) untill 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.



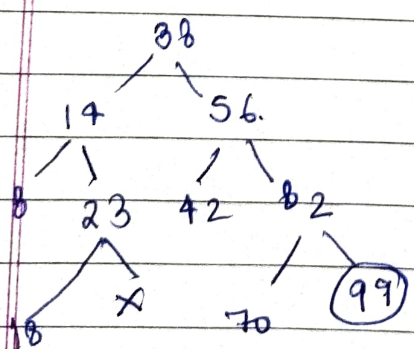
item = 23

a) 23 < 38
left side

b) 23 < 14 or 23 > 14 ✓
right child

c) 23 = 23 (search successful)

To insert 99.



item = 20.

a) 20 < 38
left

b) 20 < 14 or 20 > 14 ✓
right

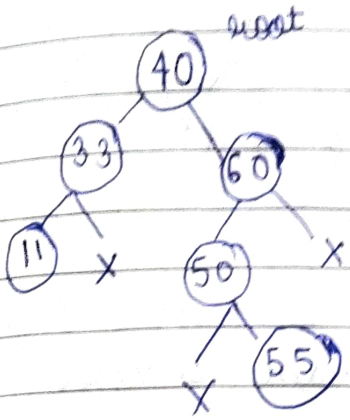
c) 20 < 23 left

d) 20 > 18 right

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

Q. Insert into empty BST in order.

40, 60, 50, 33, 55, 11

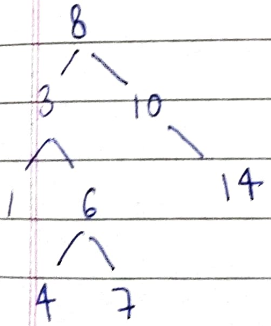


(76)

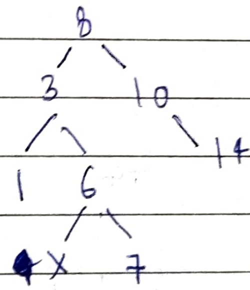
Deletion in BST

Case 1: Node is a leaf node.

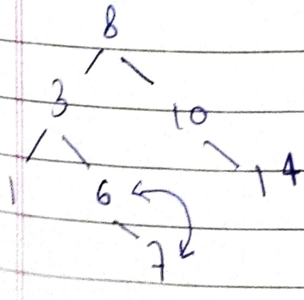
Node if a leaf node is simply removed



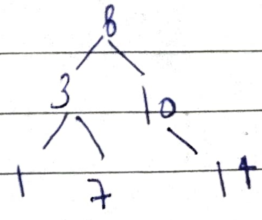
delete 4 →



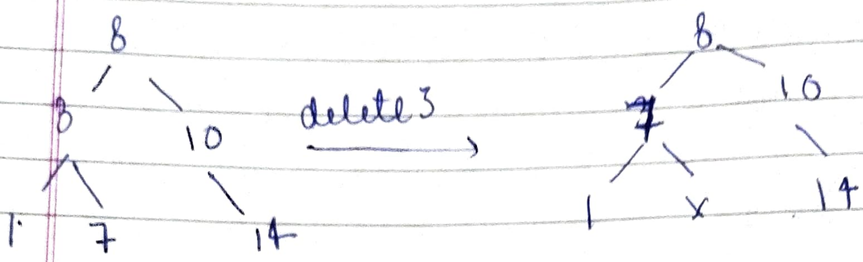
Case 2: Node has a single child.



delete 6. →



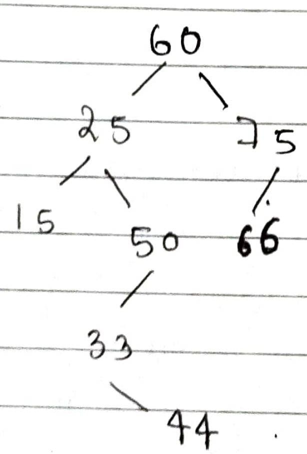
Case 3: Node having 2 children.

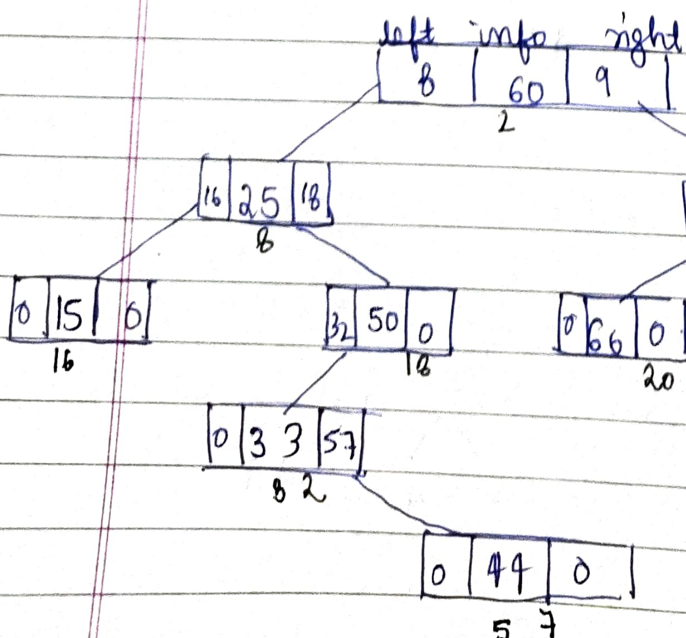


We will find the inorder successor & replace the element by its inorder successor.

Inorder : 1 3 7 8 10 14

inorder successor.





delete 44

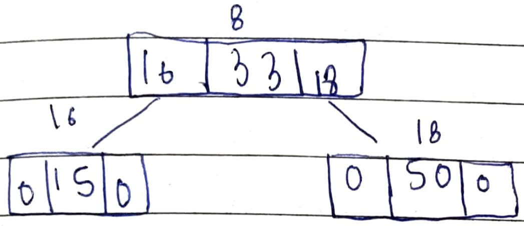
| 0 | 33 | 0 |

delete 77

| 8 | 60 | 20 |

delete 25.

inorder: 15 25 33 50 60 66



77

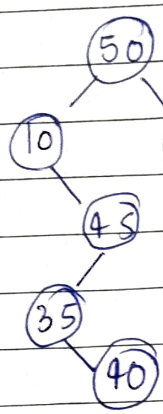
AVL trees

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

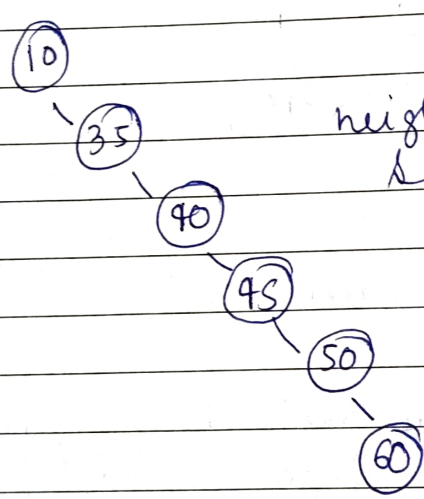
1) 50, 10, 45, 60, 35, 40.



height = 4

Search → O (log N)

10, 35, 40, 45, 50, 60

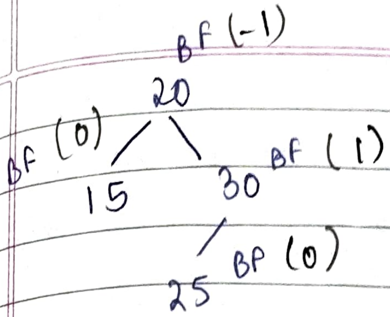


height = 5.
Search = O(n) → max

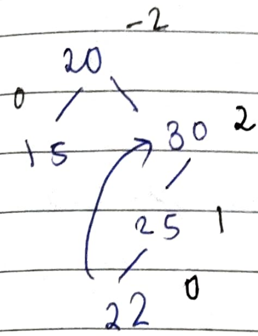
Tree is not height balanced. AVL tree can help in this.

Rotations

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

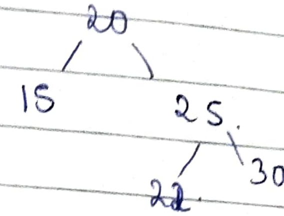


Add 22.



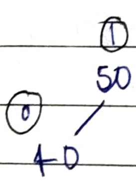
X NOT AVL but BST

$\xrightarrow{LL}$

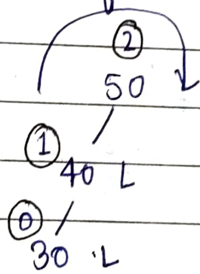


$\longleftrightarrow$ we will use rotations to amplify this case.

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

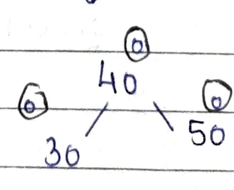


Insert 30 $\longrightarrow$

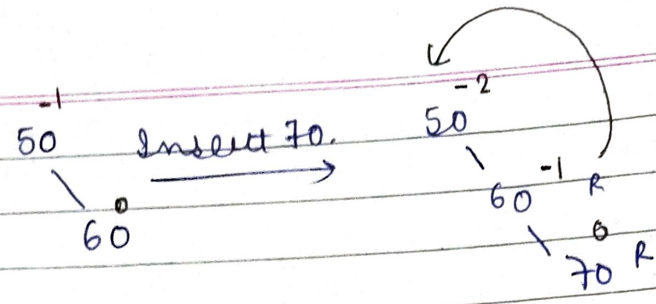


Now we can apply LL rotation
So we apply single right rotation

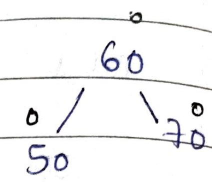
$\downarrow$



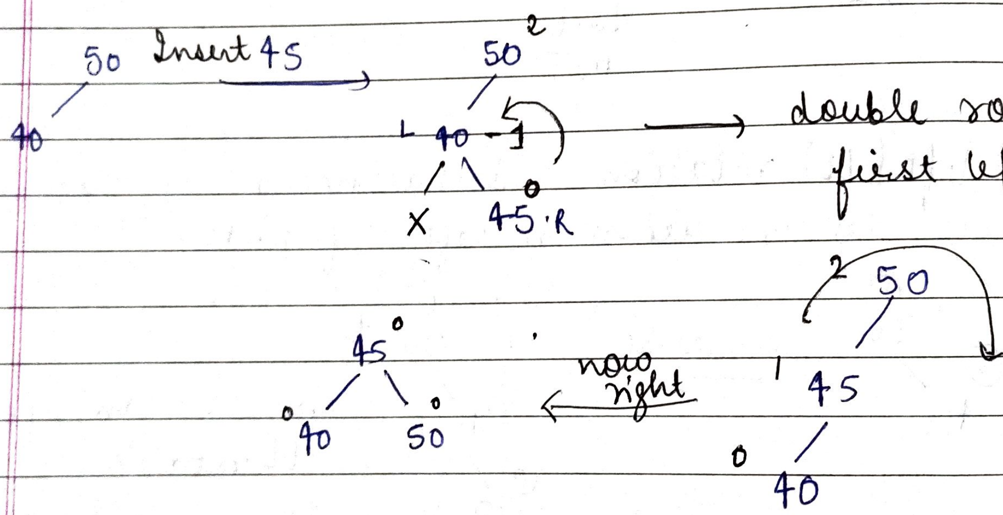
2) Right Right rotation: Imbalancing is caused due to insertion in right right node.



→ So we will rotate in left side

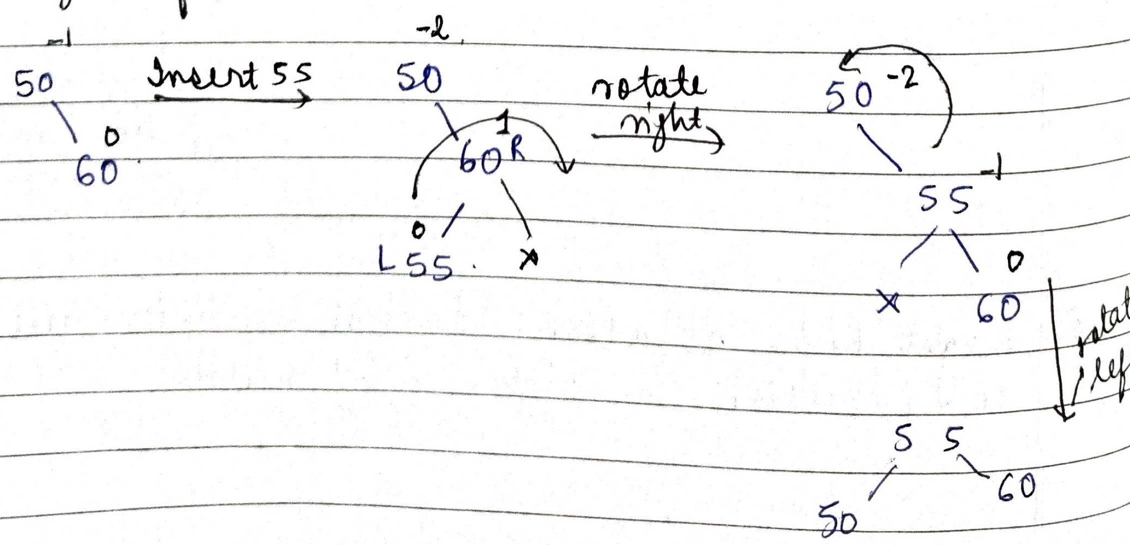


3) Left-Right rotation: Imbalancing caused due to insertion in left then right of the node.



double rotation
first left rotate

4) Right left rotation.

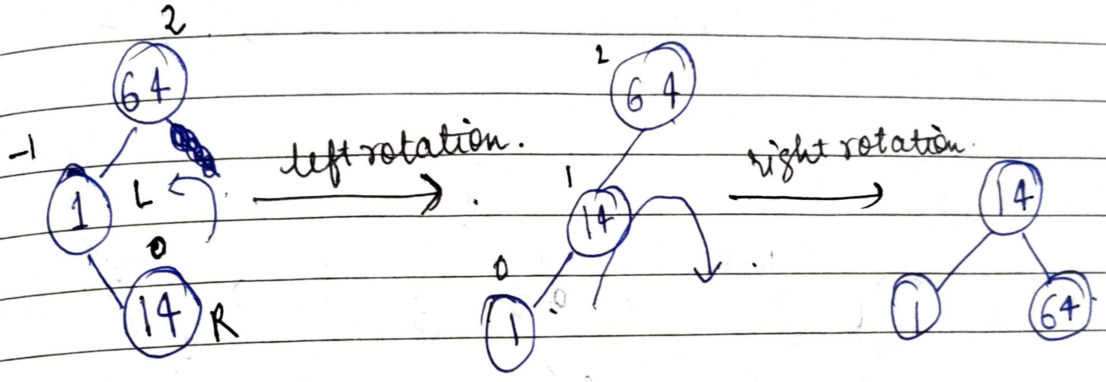


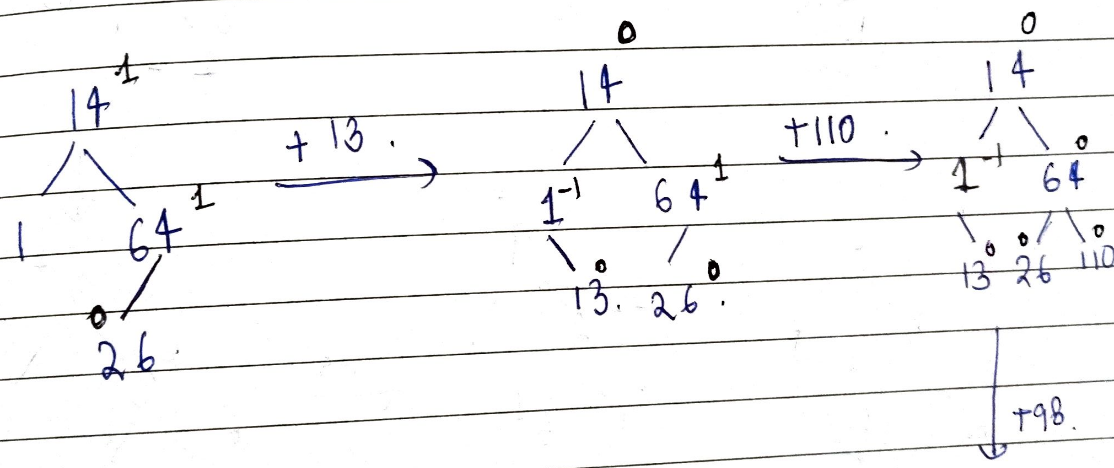
78

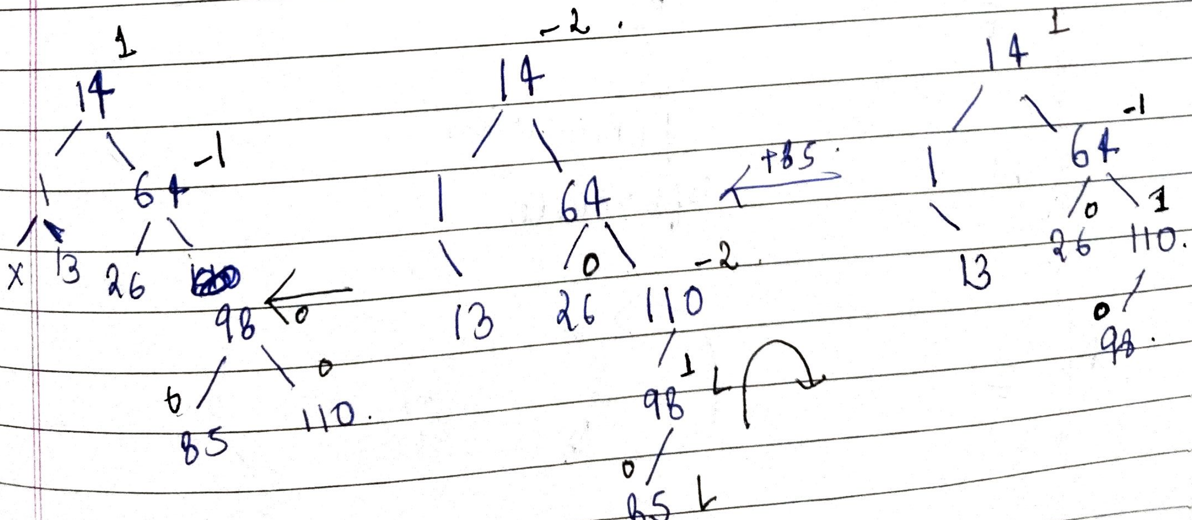
AVL tree insertion

Construct AVL search tree by inserting the following element.

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





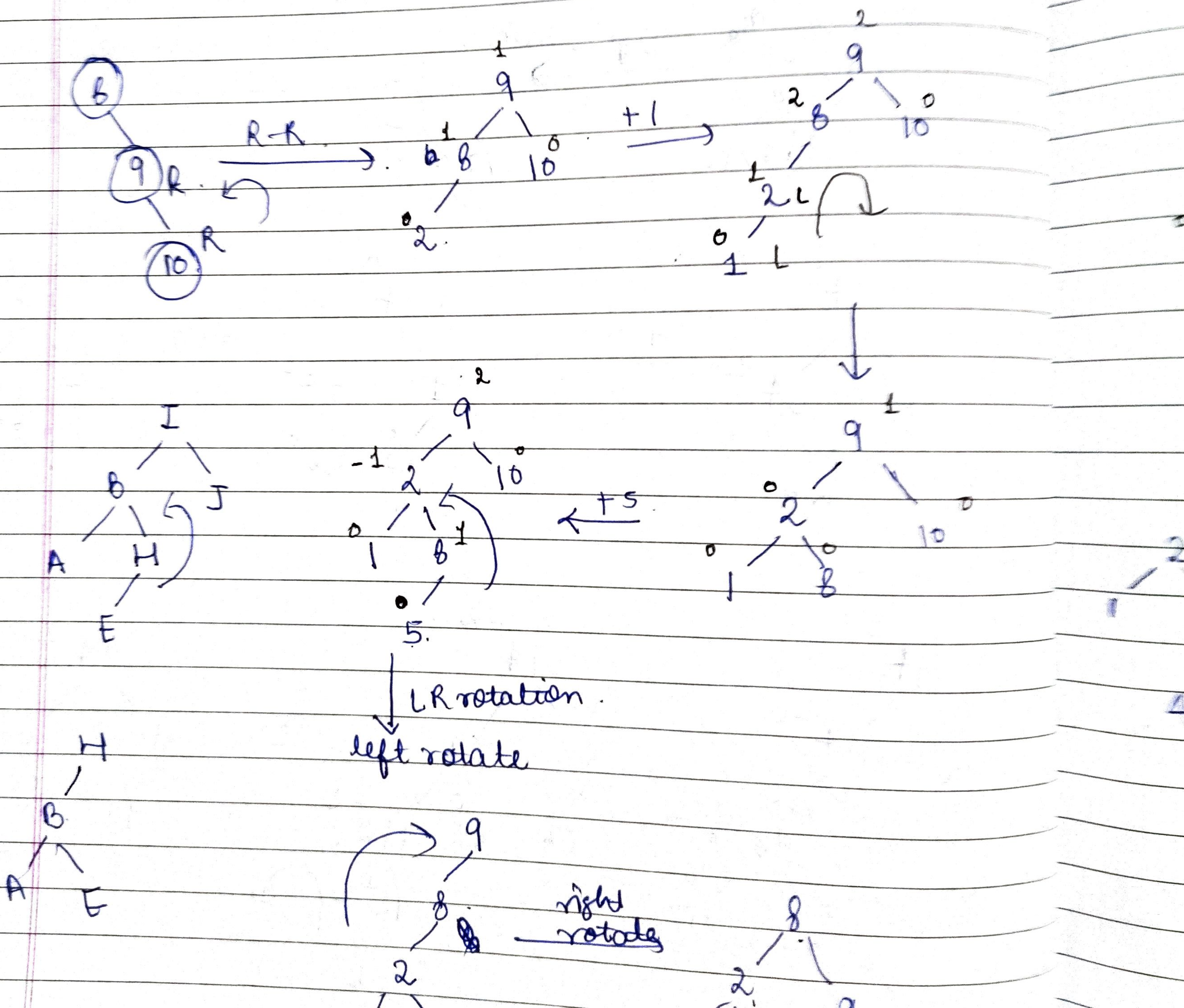


Q. Construct AVL having following key.

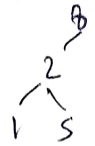
H, I, J, B, A, E, C, F, D, G, K, L

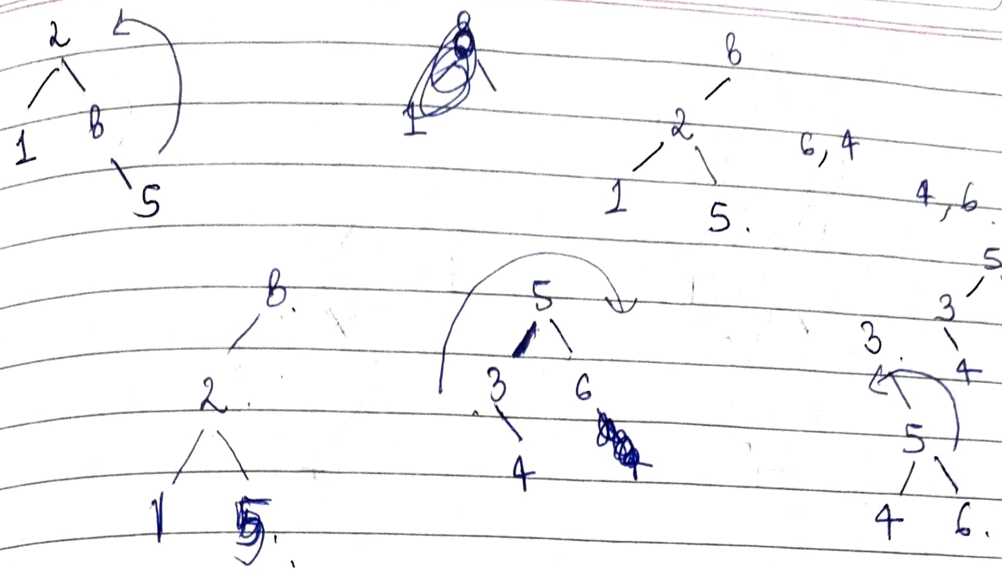
sorted order: A B C D E F G H I J K L
1 2 3 4 5 6 7 8 9 10 11 12

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

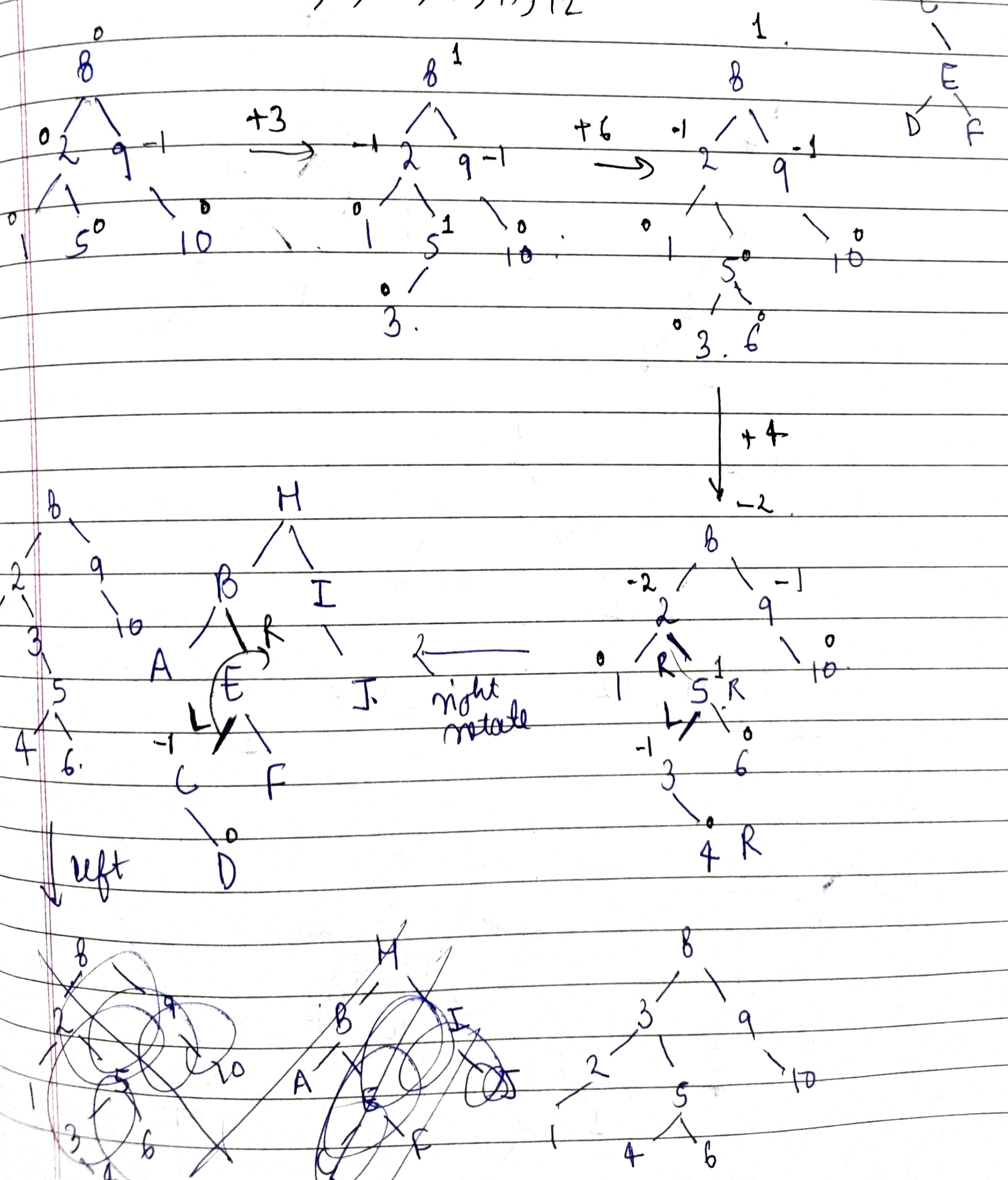


SPS
Date: / /
Page No.

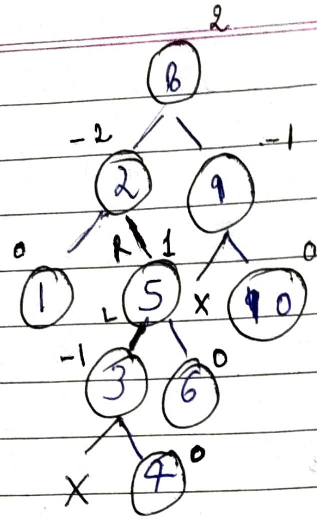




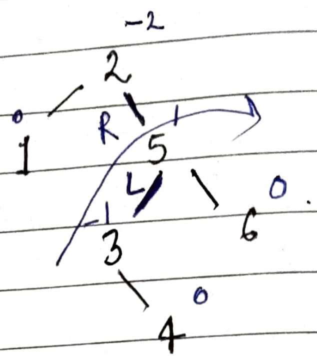
3, 6, 4, 7, 11, 12

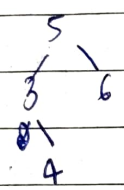


Date: / / SPS
Page No.

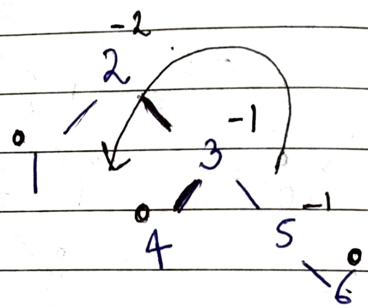


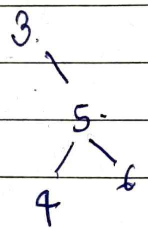
1-3



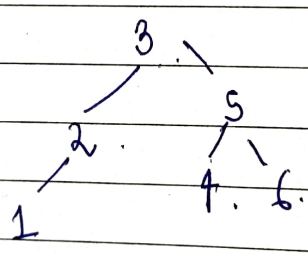


right



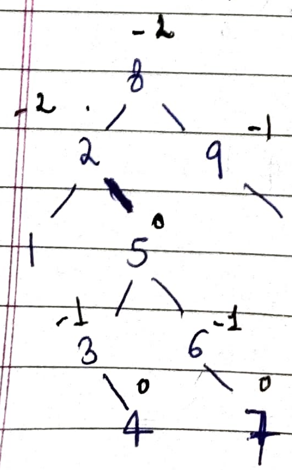


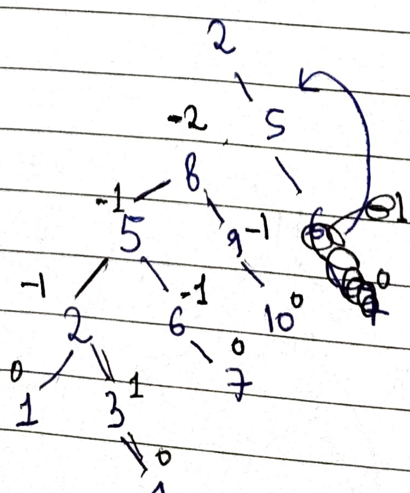
~~4~~ 1

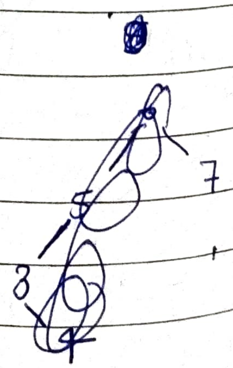


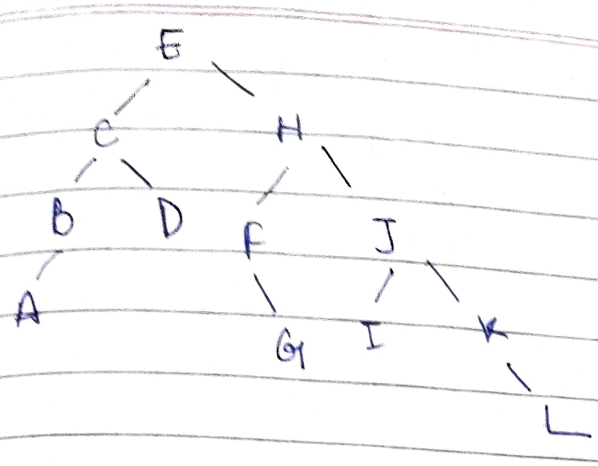
7, 11, 12

~~2~~, ~~1~~, ~~8~~, ~~3~~, ~~6~~, 4, ~~7~~









AVL tree deletion

80

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

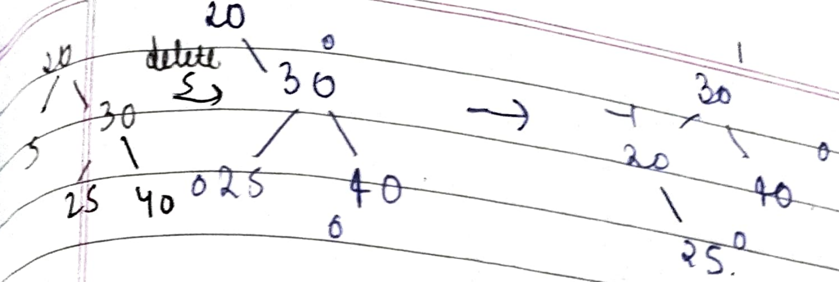
Case 1: Deletion from right sub tree

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

Case 2: Deletion from left sub-tree

i) if $BF(N) = -2$ and $BF(N \rightarrow RC) = -1$, then RR $rotat^n$.
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

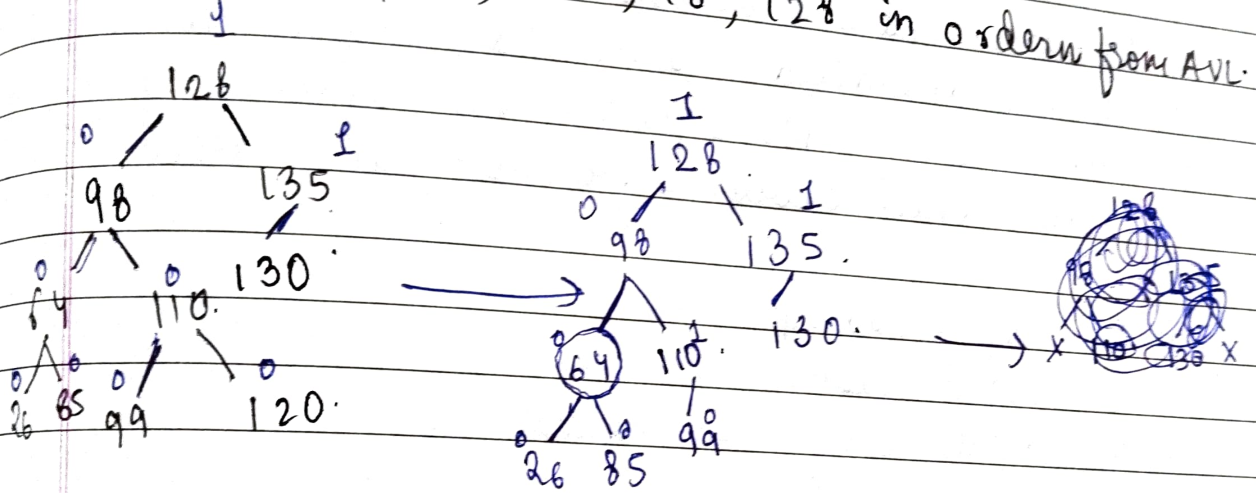
Date:
Page No.



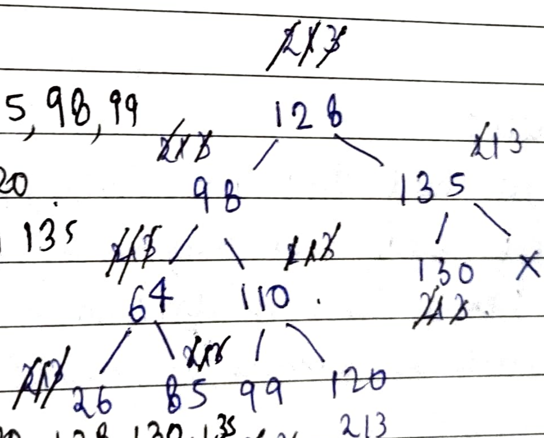
81

Example of AVL tree deletion

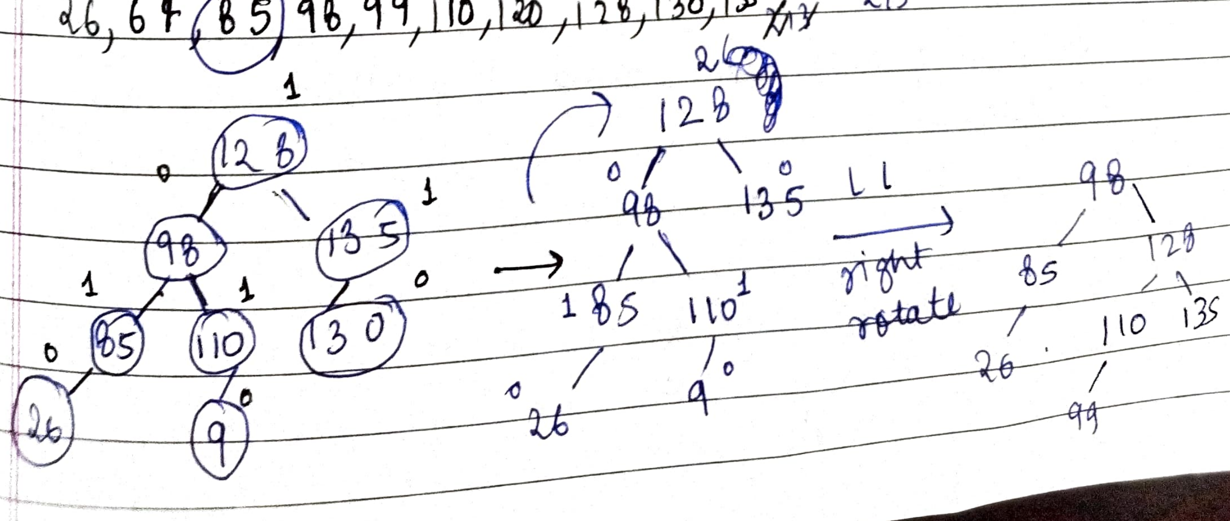
Delete 120, 64, 130, 98, 128 in order from AVL.

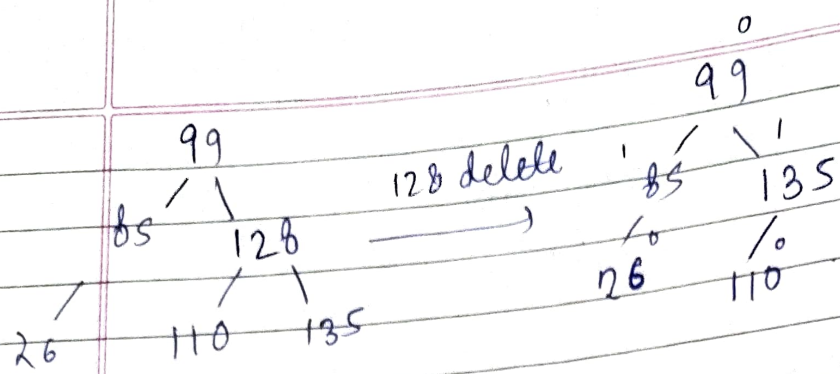


Inorder: 26, 64, 85, 98, 99,
110, 120,
128, 130 135



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





(82)

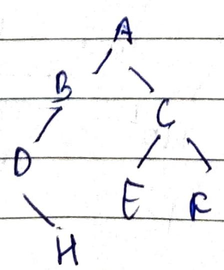
Threaded Binary Tree.

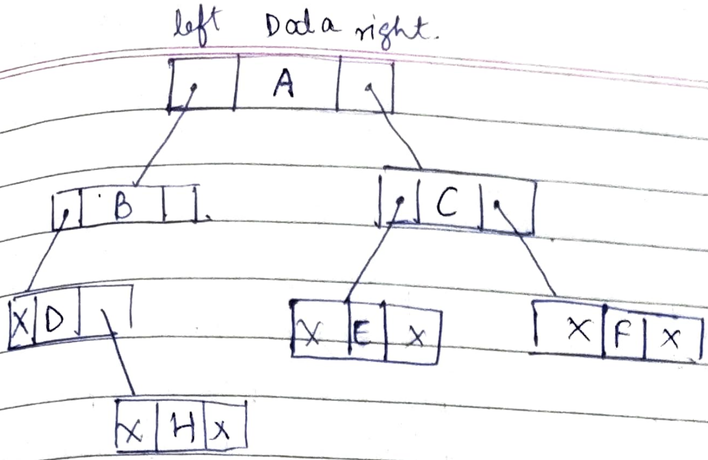
AJ Perlis and C Thornton have proposed a new binary tree called "Threaded Binary tree", which make use of NULL pointer by refrences of other node. These extra refrences are called "Thread

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.

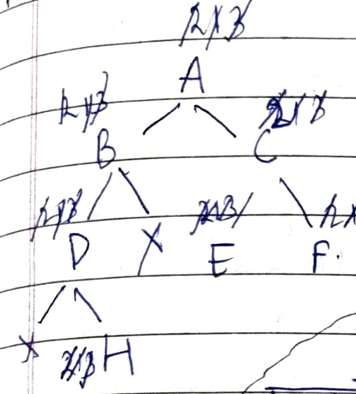




N = 7

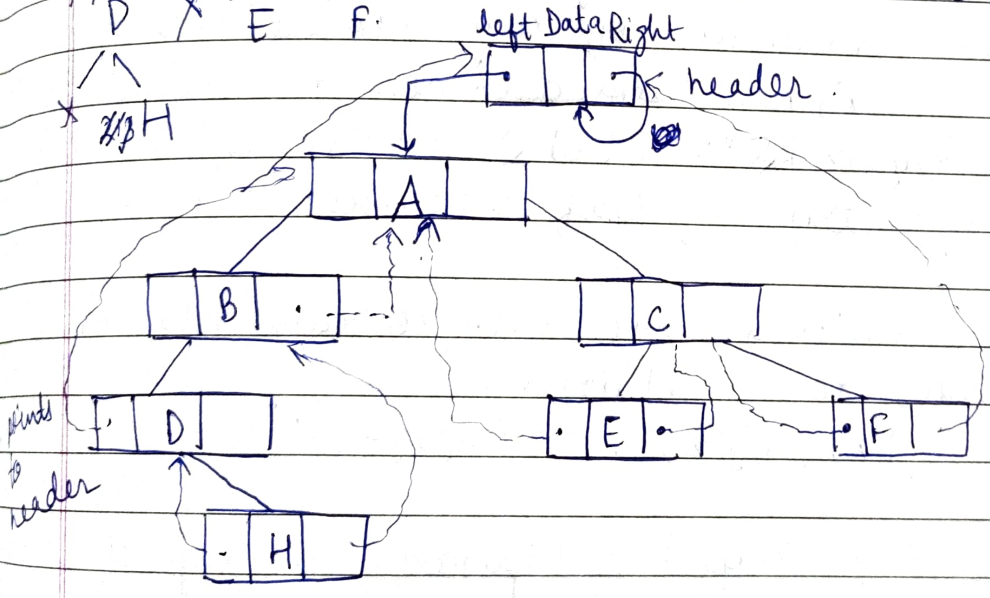
2N = 14 refrence

N+1 unused = 8



Inorder: DHBAECF

Inorder: DHBAECF.

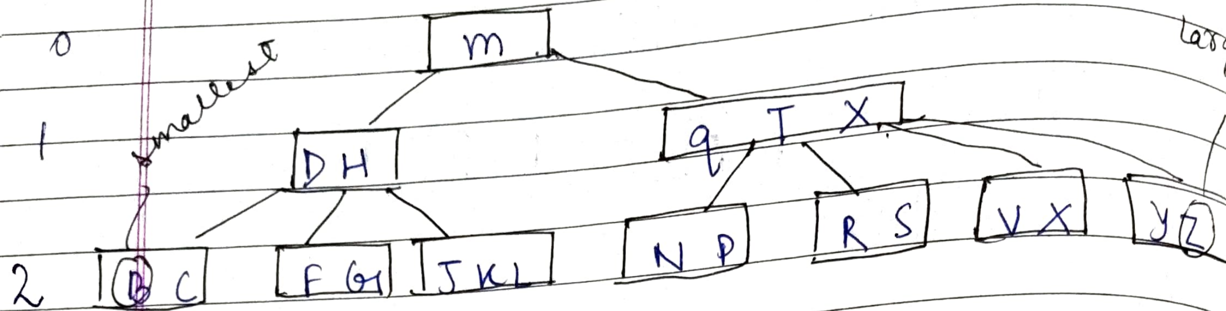


83

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.

to thousand.



If internal node x contain n[x] key then x 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
$key_1[x] \le key_2[x] \le \dots key_n[x]$.
c) leaf x = 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 $key_i[x]$ separate, the range of keys stored in sub tree
$K_1 \le key_1[x] \le K_2 \le key_2[x] \dots K_n[x] \le K_{n[x]}$

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^m. degree.

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

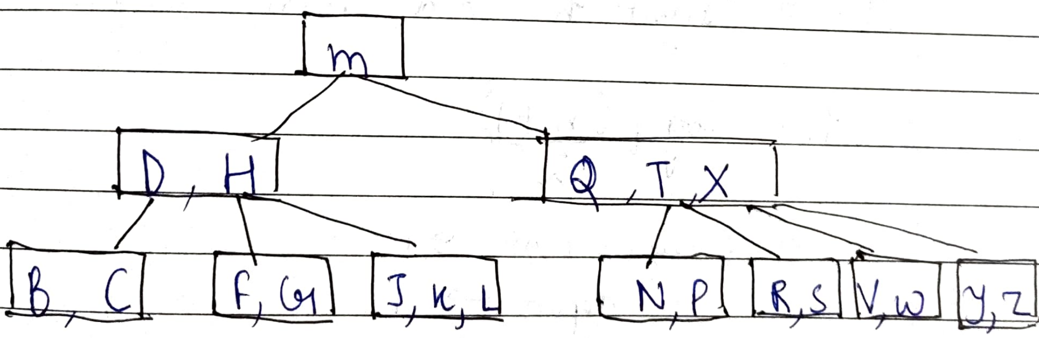
Upper bound: Every node contain at most $2t-1$ key and $2t$ children.

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)

```
i ← 1
while i ≤ n[x] & K > key_i[x]
      do i ← i+1
if i ≤ n[x] & K = key_i[x]
      then return [x, i]
if leaf [x]
      then return NIL
else DISK-READ (C_i [x])
      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 $t \geq 2$

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

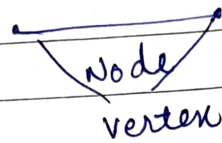
no of key on root → 1
depth → 1

90

Graph in Datastructure

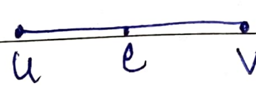
A Graph can be defined as group of vertices & edges that are used to connect those vertices

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



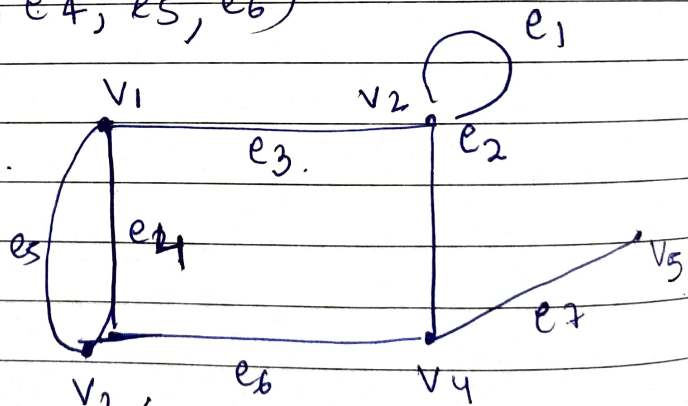
V – set of vertex /element called node
E → set of edges s.t each edge in E is unique pair of vertices [u, v]

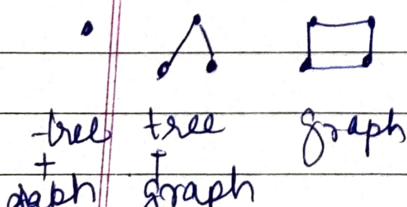
$e = [u, v]$



$G = V, E$
$V = \{ v_1, v_2, v_3, v_4, v_5 \}$
$E = \{ e_1, e_2, e_3, e_4, e_5, e_6)$

Tree is always graph.
All graph is not tree





Date: / /
Page No.

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.

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

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.

$deg(v_2) = 4$

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

Date: / /
Page No.

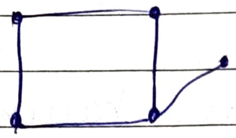
Parallel- Two edge having same vertex pair.

$e_2, e_5 = [v_1, v_3]$

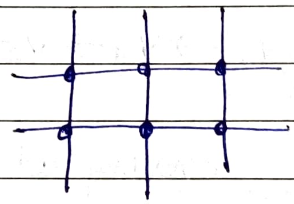
91

Types of Graph.

1) Finite graph → No of edges & vertex are countable



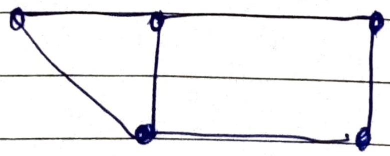
2) Infinite graph → No of edges & vertex are not countable



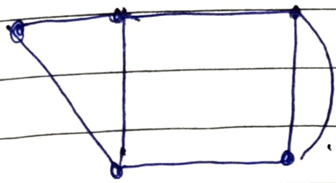
3) Trivial graph → Single node with no edges.

degree = 0

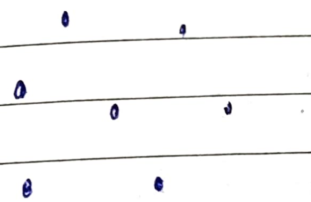
4) Simple graph → Graph having no parallel edge or self loop.



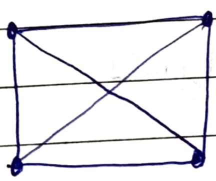
5. Multigraph - A graph having parallel edge but no self loop.



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



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

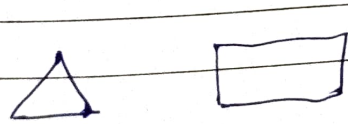


$n-1 = 3.$

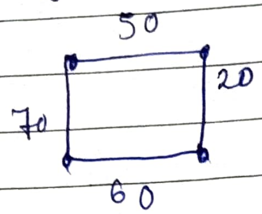
no of edges $= \frac{n(n-1)}{2}.$

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

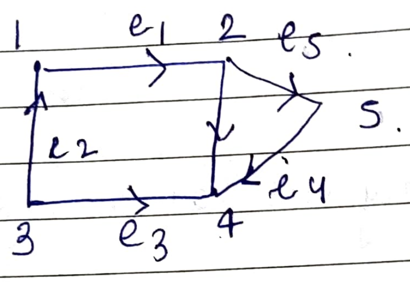
9. Regular graph - Every node has a same degree.



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



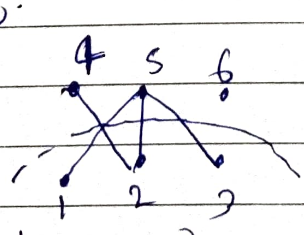
11. Directed graph: A graph having directions in edges.



Degree here are of 2 types.

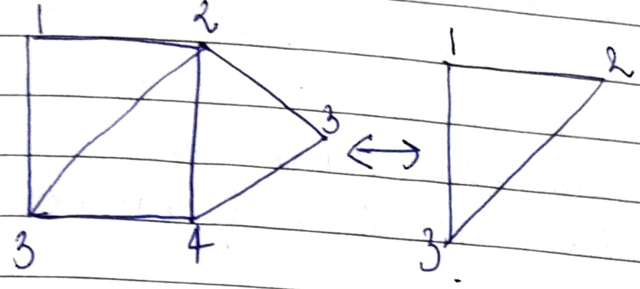
For node 2 Indegree - 1
Out degree - 2.

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



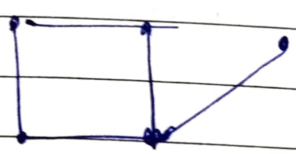
$V' = \{1, 2, 3\}$
$V'' = \{4, 5, 6\}$

13. Sub-graph - A part of graph

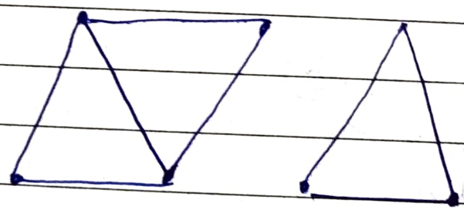


14. Connected / disconnected graph

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



When 2 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 acheived by adjacency matrix. In this representation we have to construct n*n matrix where n is vertex.

If there is edge from vertex i to j then coresponding element of matrix.

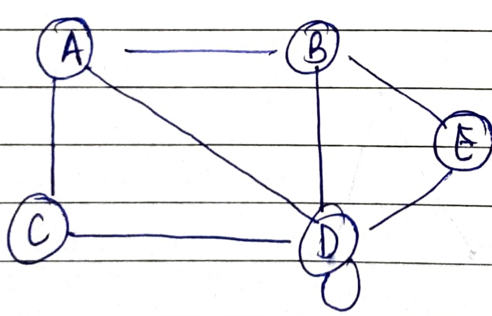
$A_{ij} = '1'$ otherwise $A_{ij} = 0$

or

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

Types of graph

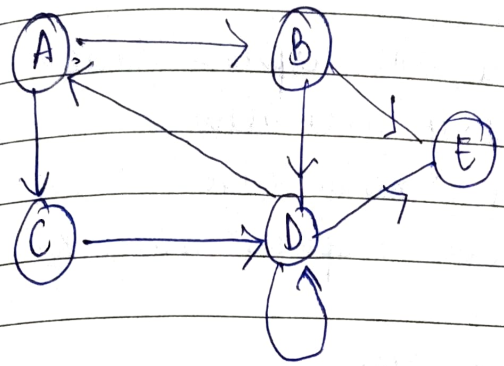
1) undirected graph



$$A = \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

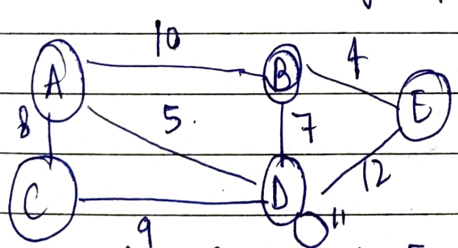
adjacency matrix

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



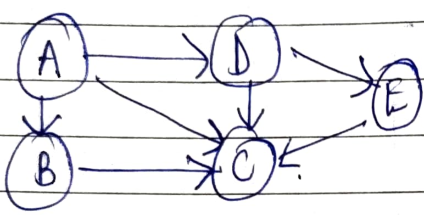
	A	B	C	D	E
A	0	10	8	5	0
B	10	0	0	7	4
C	8	0	0	9	0
D	5	7	9	11	12
E	0	4	0	12	0

(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.



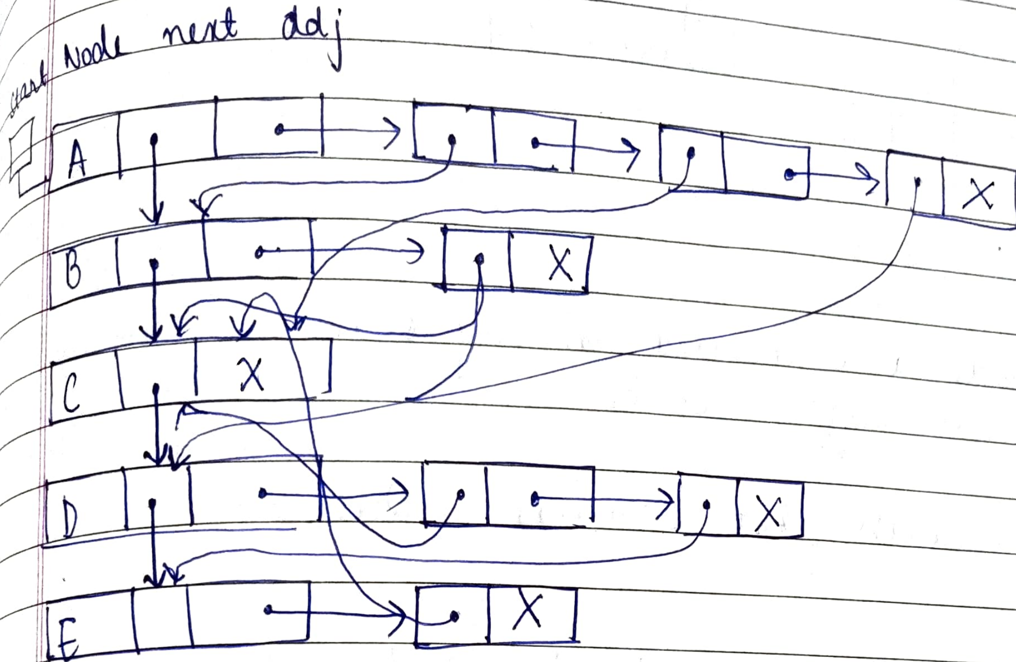
Adjacency list of graph.

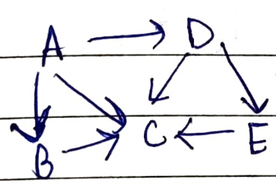
node	Adjacency list
A	D, C, B
B	C
C	—
D	C, E
E	C

Node list | node | next | Adj | → degree (optional)

Edge list | dest | link |

Node list



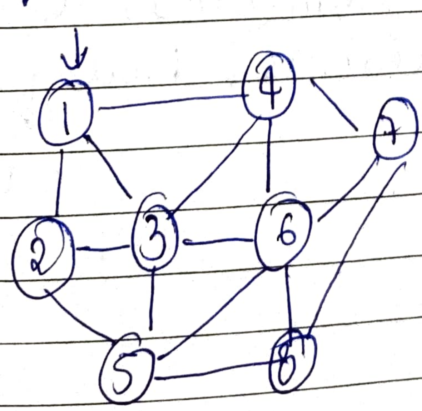


94

Graph traversing

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

DFS



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

O/p : 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)

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

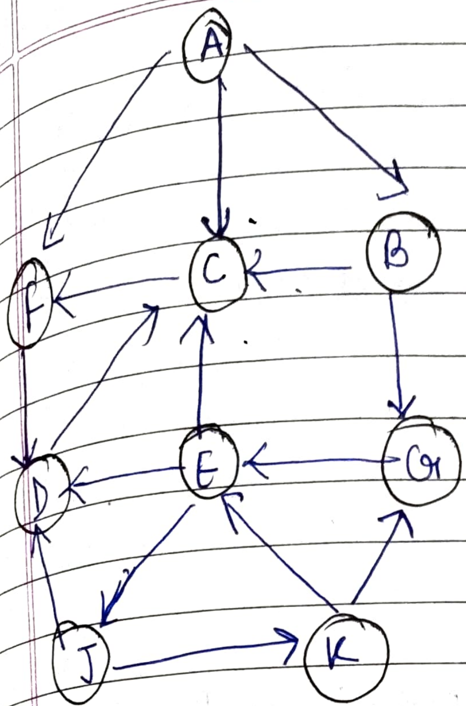
$F_1 R_1$ 1 R_2 2 F_2 3 F_3 F_4 4 R_3 F_5 5 R_4 6 F_6 7 R_5 8 R_6

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

95

DFS | BFS solved

1. Find minm path from (A to J) (BFS) → queue
2. Print all the reachable Node from J
↓
DFS → 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	C, E
8.	J.	D, K
9.	K	E, G.

A	F	C	B	G	E	J	K		
1	2	3	4	5	6	7	8	9	10

stop

Visited : A F C B D G E J 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

Date: / /

Page No.

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

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.

Lets 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 (arr[i]==x)
          return i;
  return -1
```

(2)

Introduction to Reccurence Reln

When an algorithm contain a recursive call to itself its running time can be describe by reccurence reln.

eg: Recurrence reln of Merge sort

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

Date: / /
Page No.

1) void fun(int n) = T_n
{
if (n>0) — 1
{
print (n) — 1
fun(n-1) — $T(n-1)$
}

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

$n = 0$
$T(0) = c + T(0)$

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

2) void A(n) — $T(n)$
{
if (n>0) — 1
{
for (i=0; i<n; i++) — $n+1$
{
print(n) — n
}
A(n-1) — $T(n-1)$
}.

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

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

3) fib(n)

if (n ≤ 1) —— $T(n)$

return 1 —— 1

else

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

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

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

3

Solving Reccurrence relation

There are 3 method to solve Reccurrence

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

1. Substitution method.

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

Date: / /
Page No.

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

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

$n=n-1$

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

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

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

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

$$n-k=0$$
$$n=k$$

$$T(n) = T(0) + n^2$$
$$= 1 + n^2$$
$$= O(n^2)$$

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

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

$n = n/2$ $\qquad\qquad$ $n = n/4$

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

$$\cancel{T(n) = 2\times\left[2T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+\frac{n}{2}}$$
$$\cancel{= 2\times\left[2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right]+\frac{n}{2}}$$

or

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

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

$$\vdots$$

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

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

$$n = 2^k$$

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

$$k = \log n$$

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

$$= 2^k + kn$$

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

$$= n + n \log n$$

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

Recursion tree method

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

[illegible]

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

$T(n)$ — n

|

$T(n-1)$ — $(n-1)$

|

$T(n-2)$ — $n-2$

|

$T(n-3)$ — $n-3$

⋮

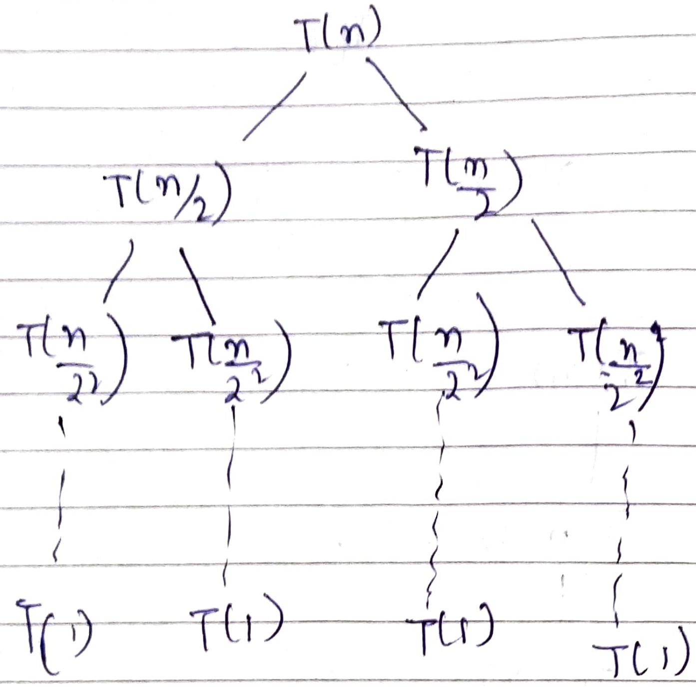
$T(0)$ — 1

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

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

$$= O(n^2)$$

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



n

n

n

n

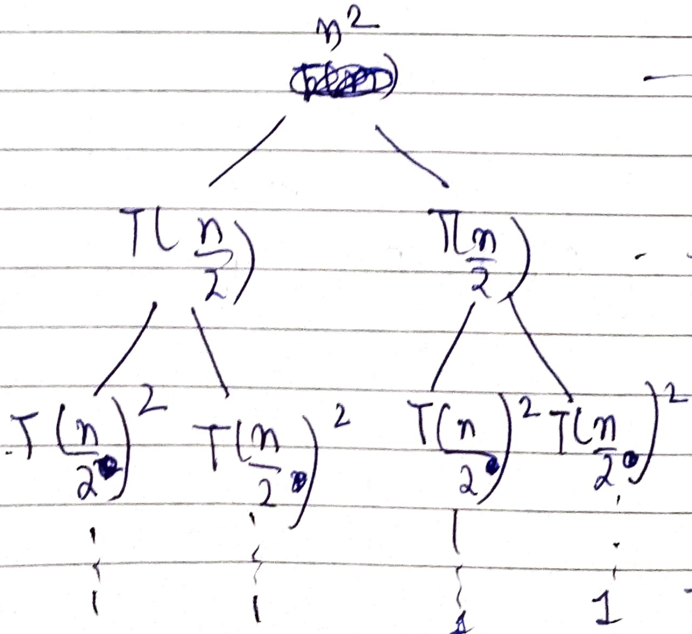
$T(n)\ kn$

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

$k = \log n$

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

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



n^2

$\frac{2n^2}{4} = \frac{1}{2}n^2$

$\frac{4n^2}{16} = \frac{1}{4}n^2$

n^2

Date: / /
Page No.

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

$\cancel{k = \log n}$.

$\cancel{T(n) = n^2 k}$

$\cancel{= n^2 \log n}$

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

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

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

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

Master Method (Divide & conquer)

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

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

where $a \geq 1$,
$b > 1$
$f(n)$ is asymptotically positive function
$\frac{n}{b}$ means $\left\lfloor \frac{n}{b} \right\rfloor$ or

$\left\lceil \frac{n}{b} \right\rceil$

Date: / / Page No.

T(n) can be bounded asymptotically as follows

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

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

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

$T(n) = 9T\left(\frac{n}{3}\right) + n$ $\qquad$ $aT\left(\frac{n}{b}\right) + f(n)$

$a = 9 \quad b = 3$

$n^{\log_a b} = n^{\log_3 9}$

$= n^2$

$f(n) = n \qquad n^{\log_a b} = n^2$

$f(n) < n^2$

then $T(n) = \Theta(n^{\log_b a})$
$= \Theta(n^2)$

Date: / /

Page No.

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

$a = 1$

$b = 3$

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

$= n^0$

$= 1$

$f(n) = 1$

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

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

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

$T(n) = \log n$

Q. $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$a = 3 \quad b = 4$

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

$f(n) = n\log n$

$f(n) > n\log n$

$T(n) = \theta(f(n))$

$= O(n\log n)$

Solving Recurrences.

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

$a = 1$ $\quad$ $f(n) = O(1) = 1$

$b = 2$

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

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

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

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

$= \log n$

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

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

$n^{\log_b a} = n^{\log_1 1} = n^0 = 1$

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

$T(n) = \Theta(f(n))$

$= \Theta(n^3)$

Date / /
Page No

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

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

Compare 2 adjacent elements a & b
if a > b then swap a and b.

5	1	6	2	4	3

$n-1$

1 5 6 2 4 3.

1 5 2 6 4 3.

1 5 2 4 6 3.

1 5 2 4 3 6.

1 5 2 4 36

1 2 5 4 36

1 2 4 5 36

1 2 4 3 5 6

1 2 4 3 5 6.

1 2 4 3 56

1 2 3 4 56.

Pass 1 → $n-1$
Pass 2 = $n-2$

Bubble sort (A, n)

```
for(i=n; i>=1; i--)
    for(j=1; j<=i-1; j++)
        if a[j] > a[j+1]
            swap(a[j], a[j+1])
```

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

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

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

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

$$= O(n^2)$$

Time

Worst case = $O(n^2)$

Best case = $O(n)$

Avg case = $O(n^2)$

Space

$O(1)$ while swapping temp variable