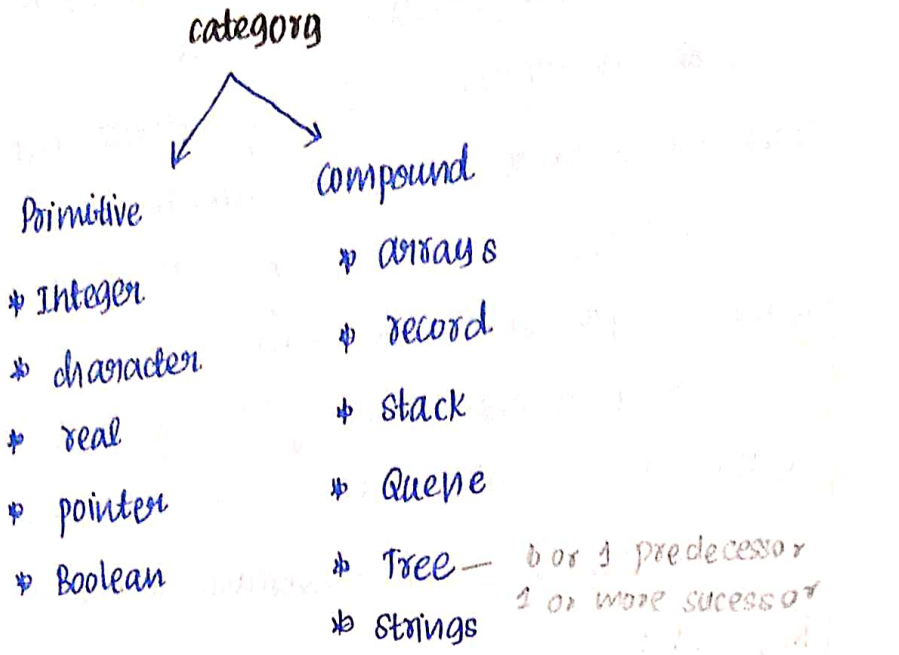
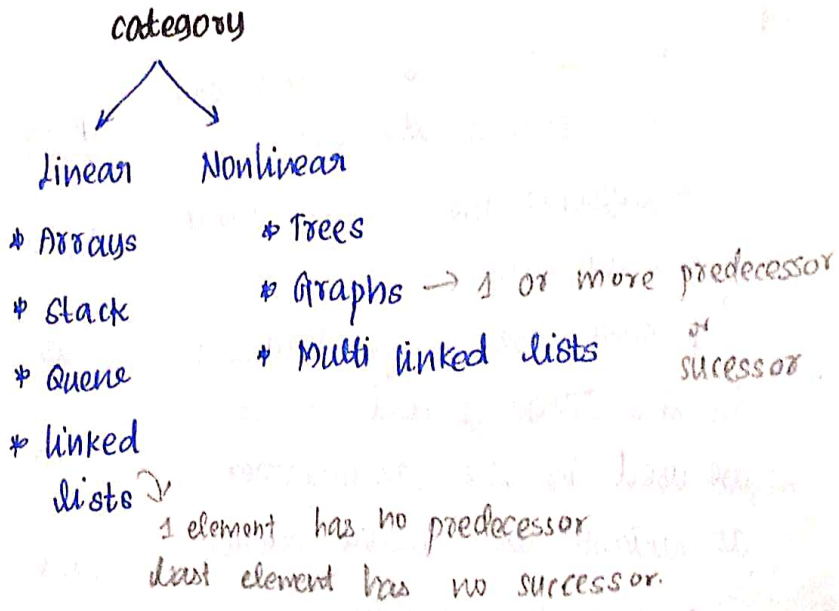
CLASSIFICATION OF DATA TYPES :-





Abstract Data Type (ADT) :-

Specification of Data structures.
Data object
operations on the data object. (ADT)
Application ⇔ specification ⇔ Implementation
(User)

Array (ADT) :-

Creation (Name, size) ↔ pointer
Store (index, element)
Retrieve (index) → return value (element)

Pointer → Create (size)

1. Allocate memory to hold size elements
2. Return the base address

store (index, element)

1. Arrayname [index] = element
2. Return.

Retrieve (index)

1. Return Array Name [index]

Linear lists :-

List ADT

$l = (e_1, e_2, e_3 \ldots e_n)$

ordered collection of finite no. of elements

e_1 is the first element

e_n is the last element of the list.

operations :-

- isEmpty () : Returns true iff the list has 0 elements.
  otherwise false.
- insert (index, element) : inserting the given element in the given index position increasing the length of the list by 1.
- Length () : Returns the no. of elements in the list.
- Delete (index) : Deleting the element in the given index position.
- index of (element) : Returns the index position of the given element.

Conventions :-

Algorithm Name (parameter list)

// - - - - -

↓

(comments for understanding purpose)

Ex:-

Algorithm for insert (insert, element)

// Inserting the given 'element' in the given 'index'

// position. The 'index' should be less than or equal to

// length of the list +1.

1.

After 2.

comment each step is numbered

Indentation is used to specify block structure

'in the given index'

Assignment :-

Var = expression

Eg :-

Var = $\lfloor x/2 \rfloor$ → floor

mid = $\frac{(low + high)}{2}$. $\lfloor (low + high)/2 \rfloor$

I/O : Read Var list

Print value / Expression.

If statement :-

if (condition)

→ statement

→ -

→ -

[else

=

=

-]

[end if]

While statement:- [only while statement we have]

while (condition)

statement

—

—

—

[end while]

for statement:

for var = initial value {to / down to} final value step step value (optional if it is 1 or -1)

Not → statement start

statement

—

—

—

[end for]

If it is not returning value,

var = Name of function (actual parameters).

n = L. length()

If return,

var/expression.

For memory allocation,

var = get Node (Node structure)

Temp = get Node (NODE)

NODE

Data	link

Data (temp)

Link (temp)

Eg:-

```
Struct node
{
int data;
Struct node * link;
};
struct node * s;
```

Declaration of Structure in implementation point of view:

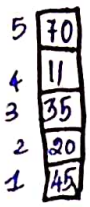
S = (struct node *) malloc (size of (struct node));

STACK :

Last in First out.

Array n = 5

Stack Empty → Top = 0



STACK OPERATION :

ISFULL (S), PUSH (S, x), ISEMPTY (S), POP(S), PEEK(S)

Algorithm ISFULL(S)

1. If S.top ≥ S.length
2. Return True.
3. Else
4. Return False.

Algorithm PUSH (S, x)

1. If ISFULL (S)
2. print "stack is full. Can't insert"
3. Exit
4. s.top = s.top + 1
5. S[s.top] = x
6. Return.

Algorithm ISEMPTY (S)

1. If S.top ≤ 0
2. Return True
3. Else

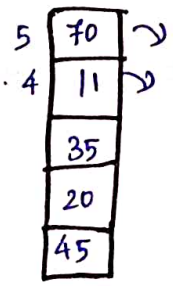
Return False

Algorithm POP (S)

1. If ISEMPTY (S)
2. print "stack is Empty . Can't delete".
3. Exit
4. x = S[S.top]
5. S. top = S.top - 1
6. Return x.

stack Empty →

Top = ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~



Algorithm PEEK (S)

1. IF ISEMPTY(S)
2. print "stack is Empty"
3. Exit.
4. Return S[S.top]

APPLICATIONS OF STACK:-

① Infix to postfix Conversion :-

Infix Expression :

Write an operator between the operands is known as Infix Expression.

Ex: a+b, a+b/c .

postfix Expression :

Writen an operator after the operands is known as postfix Expression.

Ex: ab+, abc/+.

Infix Expression example,

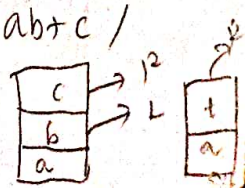
a+b, a+ b/c,

Convert an Infix (a+b)/c expression into another expression is known as postfix Expression.

ab+, a bc/+, a+

(a+(b/c))

ab+c/



a + b/c - d*e

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

a b c / + d e * -

operator precedence,

1. /
2. *
3. +
4. −

(x-y) / (z+w) - p * q / r

(((x-y) / (z+w)) - ((p*q)/r))

xy - z w+/ p q * r/ -

^ - exponential symbol - top precedence among all the symbols.

a^b^c

Right → left

$(3^2)^4$ $(3)^{(2^4)}$

$= 9^4$ $= 3^{16}$.

a*b / (c+d)^e + g - f/h

~~((a*b) / ((c+d)^e)) + (g - (f/h))~~

~~a b * cd + ae ^ / gfh /-+~~

((((a*b) / ((c+d) ^e)) + g) - (f/h)))

ab * cd + e^/ g + fh /-

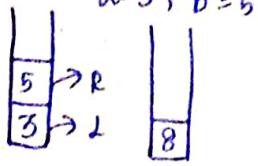
$\left(\frac{x+y}{e+r}\right)^t - \frac{1}{r}$

(((x+y) / (e+r))^t) - (1/r))

xy + er+/t^ 1r/ -

postfix Expression ab+

a=3, b=5



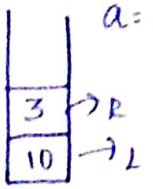
L = 3

R = 5 L+R

3+5 = 8.

postfix Expression ab- cd/+

a=10, b=3, c=6, d=2

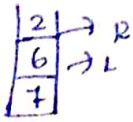


R=3

L=10

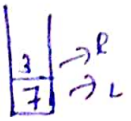
L-R

10-3 = 7



L/R

6/2 = 3



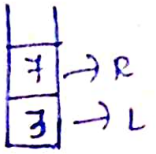
L+R

7+3=10

postfix Expression xy+ er+ /t^1r/-

x=3, e=1, t=2

y=7, r=1



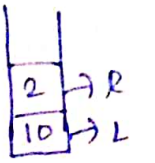
R=7

L=3

L+R

= 3+7

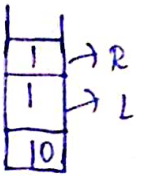
= 10



L/R

= 10/2

= 5.



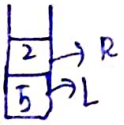
R=1

L=1

L+R

= 1+1

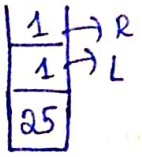
= 2



L^R

= 5^2

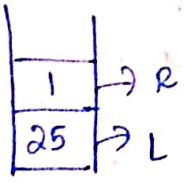
= 25



L/R

= 1/1

= 1



L-R

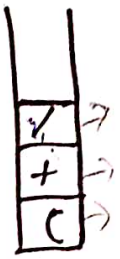
= 25-1

= 24

operator	Instack priority (ISP)	Incoming priority (ICP)	
)	-	-	
^	3	4	Operands. There is no priority
*, /	2	2	
+, -	1	1	
(	0	4	

(a+b) /c

x = i^{th} char in the infix expression



if x is operand,

append x to postfix expression

if x is (,

push it to stack.

if x is),

pop all operators from stack till '(' and append it to the postfix expression expect 'c'.

if x is a operator,

pop all operators on stack having isp higher than ICP of x and append it to the postfix expression.

push x into stack.

x	postfix
(	ab+c /
)	
/	

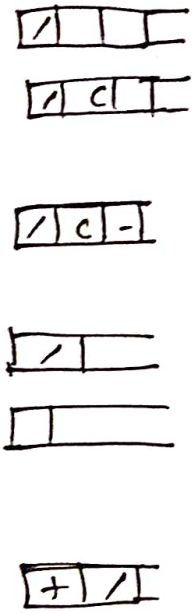
a/(b-c)+d/e

x = i^{th} char in the infix expression.

if x is operand.

append x to postfix expression

if x	Postfix
a	a
/	
(	ab
b	
-	abc
c	
)	abc-
+	abc-/
d	
/	abc-/d



Algorithm POSTFIX (E)

// E - is the infix expression

1. S = create ()
2. P = ' ' ; j = 1
3. For i = 1 to E.length
4. x = E[i]
5. If x is operand
6. P[j] = x
7. j = j+1
8. Else if x is '('
9. PUSH (S, x)
10. else if x is ')'
11. while PEEK(S) ≠ '('
12. t = POP(S)
13. P[j] = t
14. j = j+1
15. POP (S)
16. Else
17. While ISP(PEEK(S)) ≥ ICP (x)
18. t = POP (S)
19. P[j] = t
20. j = j+1
21. PUSH (S, x)
22. return

stack ADT:-

```
typedef int element;
typedef struct {
        int length;
        int top;
        Element *a;
}stack;
Void createstack (int, n);
Void push (Element x);
Element pop ();
Element peek ();
```

```
#include "stackADT.h"
#include<stdio.h>
#include<stdlib.h>

Stack s;
        int isEmpty();
        int isfull();
void createstack (int n) {
        s.a = (Element *) calloc (n, sizeof (Element));
        s.length = n;
        s.top = -1;
}
int isEmpty()
void push (Element x) {
        if (s.top <= -1)
                return 1;
        else
                return 0;
```

Arraystack.c .

```
#include "stackADT.h"
#include<stdio.h>
#include<stdlib.h>
stack s;
int isEmpty();
int isfull ();
void createstack (int n) {
```

```
    S.a = (Element*) calloc (n, sizeof Element));
    S.length = n;
    S.top = -1;
}
Int isEmpty ()
    if (S.top <= -1)
        return 1;
    else
        return 0;
}
Void push (Element x) {
        if (isFull ()) {
            printf ("\n Stack overflows. cannot insert.");
            return;
        }
        S.top = S.top + 1;
        S.a [S.top] = x;
}
Element pop() {
        Element x;
        if (isEmpty ()) {
            printf ("\n stack Underflows. cannot delete.");
            return;
        }
        x = S.a [S.top];
        S.top = S.top - 1;
        return x;
}
    Element peek () {
        Element x;
```

```
if (isEmpty()) {
    printf ("\n stack underflows. cannot delete.");
    return 0;
}
x = s.a[s.top];
return x;
}
```

Stack. Main .c

```
# include <stdio.h>
# include "Arraystack.c"
int main () {
    int op, n;
    Element x;
    printf ("\n Enter size of stack:");
    scanf ("%.", &n);
    Createstack (n);
    do {
        printf ("\n1-push 2-pop 3-peek 4-Exit");
        scanf ("%d", &op);
        switch (op) {
            case 1:
                printf ("\n Enter the value to be
                                pushed");
                scanf ("%d", &x);
                push (x);
                break;
            case 2: x = pop();
                if (x != 0) {
                printf ("\n The Elemented deleted is
                                %d", x);
            } break;
```

```
case 3:
        x = peek();
        if (x != 0){
            printf("\n The Elemented on the
                    top of stack is %d", x);
        }
        break
    }
}
while (op<=3);
return 0;
}
```

Algorithm to EVALUATE (E):.

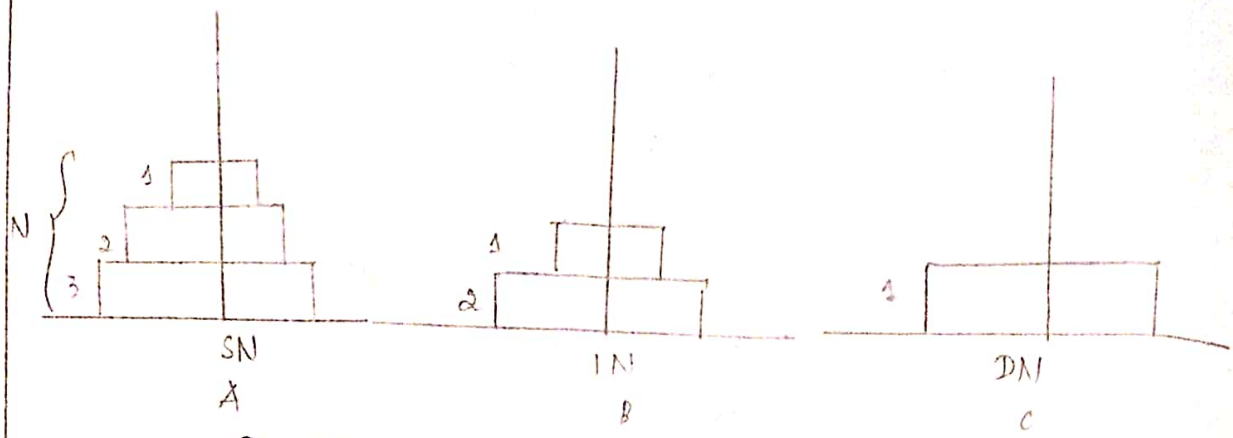
// E is postfix expression.

1. Create . stack (S)
2. For i = 1 to E. length
3. c = E[i]
4. if c is operand
5. PUSH (S, c)
6. else
7. R = POP(S)
8. L = POP(S)
9. switch (c)
10. case '+' : result = L+R
11. case '-' : result = L-R
12. case '*' : result = L*R
13. case '/' : result = L/R
14. case '^' : result = L^R
15. PUSH (S, result)
16. end for
17. Return POP (S)

Tower of Hanoi :-

N → discs

3 Needles → source Needle (A), Intermediate Needle (B), Destination Needle (C).



① Move N-1 discs from A to B
↓ ↓
SN DN

② Move disc N from A to C.

③ Move N-1 disc from B to C
SN DN

Recursive Algorithm HANOI (N, SN, IN, DN):- $[2^n - 1]$

1. If N > 0.
2. Move N-1 discs from SN to IN ⇒
   HANOI (N-1, SN, DN, IN)
3. print 'Move disc', 'N', 'from', 'SN', 'to', 'DN'.
4. Move N-1 discs from IN to DN ⇒
   HANOI (N-1, IN, SN, DN).

HANOI $(3, A, B, C)^{5}$:- Return
SN IN DN

1. 3 > 0 yes
2. HANOI (2, A, B) (SN IN DN)
   1. 2 > 0 yes.
   2. HANOI (1, A, B, C)

1. 1>0 yes.
2. HANOI (0, A, C, B)

   1. 0>0 NO
   5. Return.

3. Print 'Move disc 1 from A to C'.
4. HANOI (0, B, A, C)

   1. 0>0 NO
   5. Return

5. Return

3. Print 'Move disc 2 from A to B'
4. HANOI (1, C, A, B)

1. 1>0 yes.
2. HANOI (0, C, B, A)

   1. 0>0 NO
   5. Return

3. Print 'Move disc 1 from C to B
4. HANOI (0, A, C, B).

   1. 0>0 NO
   2. Return

5. Return.

3. Print 'Move disc 3 from A to C'.
4. HANOI (2, B, A, C)

1. 2>0. Yes.
2. HANOI (1, B, C, A)

   1. 0>0 NO
   5. Return

3. Print 'Move disc' 1 from B to A
4. HANOI (0, C, B, A)

   1. 0>0 NO
   5. Return

5. Return

3. print 'Move disc 2 from B to C'
4. HANOI (1, A, B, C)

1. 1>0
2. HANOI (0, A, C, B)
   1. 0>0 No
   5. Return
3. print 'Move disc 1 from A to C'
4. HANOI (0, B, A, C)
   1. 0>0 NO
   5. Return
5. Return

5. Return

5. Return

Output:

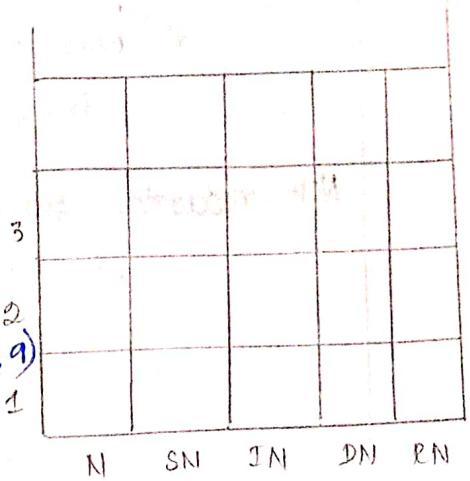
1. Move disc 1 from A to C
2. Move disc 2 from A to B
3. Move disc 1 from C to B
4. Move disc 3 from A to C
5. Move disc 1 from B to A
6. Move disc 2 from B to C
7. Move disc 1 from A to C

Non Recursive Algorithm:

N - INT
SN, IN, DN → char
RN → step No.

HANOI (N)

1. t = (N, 'A', 'B', 'C', 18)
2. PUSH (S, t)
3. t = PEEK (S)
4. If t.N = 0.
5. go to t.RA.
6. Else
7. t = (t.N-1, t.SN, t.DN, t.IN, 9)
8. go to step 2
9. t = POP(S)
10. t = PEEK(S)
11. print 'Move disc', t.N, 'from', t.SN, 'to', t.DN
12. t = PEEK(S)
13. t = (t.N-1, t.IN, t.SN, t.DN, 15)
14. goto step 2.
15. t = POP(S)
16. t = PEEK(S)
17. goto t.RA
18. return.



Algorithm Analysis :-

Time complexity → O, Ω, Θ → Order of growth

Space Complexity

Constant Complexity	$O(1)$
Logarithmic complexity	$O(\log n)$
Linear complexity	$O(n)$
nlogn Complexity	$O(n \log n)$
Quadratic Complexity	$O(n^2)$
$n^2 \log n$ complexity	$O(n^2 \log n)$

cubic complexity $O(n^3)$

polynomial complexity $O(n^d)$.

Polylogarithmic complexity $O(n^d \log n)$

$n^{d-1} \log n \propto n^d < n^d \log n$

$\log < n \propto n \log n$

Non-recursive Algorithm:

$n^d < n^d \log n < n^{d+1}$

Exponential Complexity a^n

2^n
3^n
4^n

PUSH (S, x)

1. overflow → O(1)
2. S.top = S.top + 1 } O(1)
3. S[top] = x

POP(S, x)

1. Underflow
2. x = S.a[S.top]
3. S.top = S.top - 1
4. return x.

Algorithm Linear Search (A, x)

1. For i=1 to A.length
2. if A[i] = x
3. return i
4. return -1

Matrix Addition:

For i=1 to A rows

For j= 1 to A columns.

C[i,j] = A[i, j] + B[i, j]

$O(m \times n)$ $\quad N = \max(m,n)$

$\Theta(N * N)$

$\Theta(N^2)$

Matrix Multiplication:-

```
for i=1 to A.rows
    for j=1 to B columns
        C[i,j]=0
    for k=1 to A.columns.
C[i,j]= C[i,j] + A[i,k] * B[k,j].
```

$m * n * p$

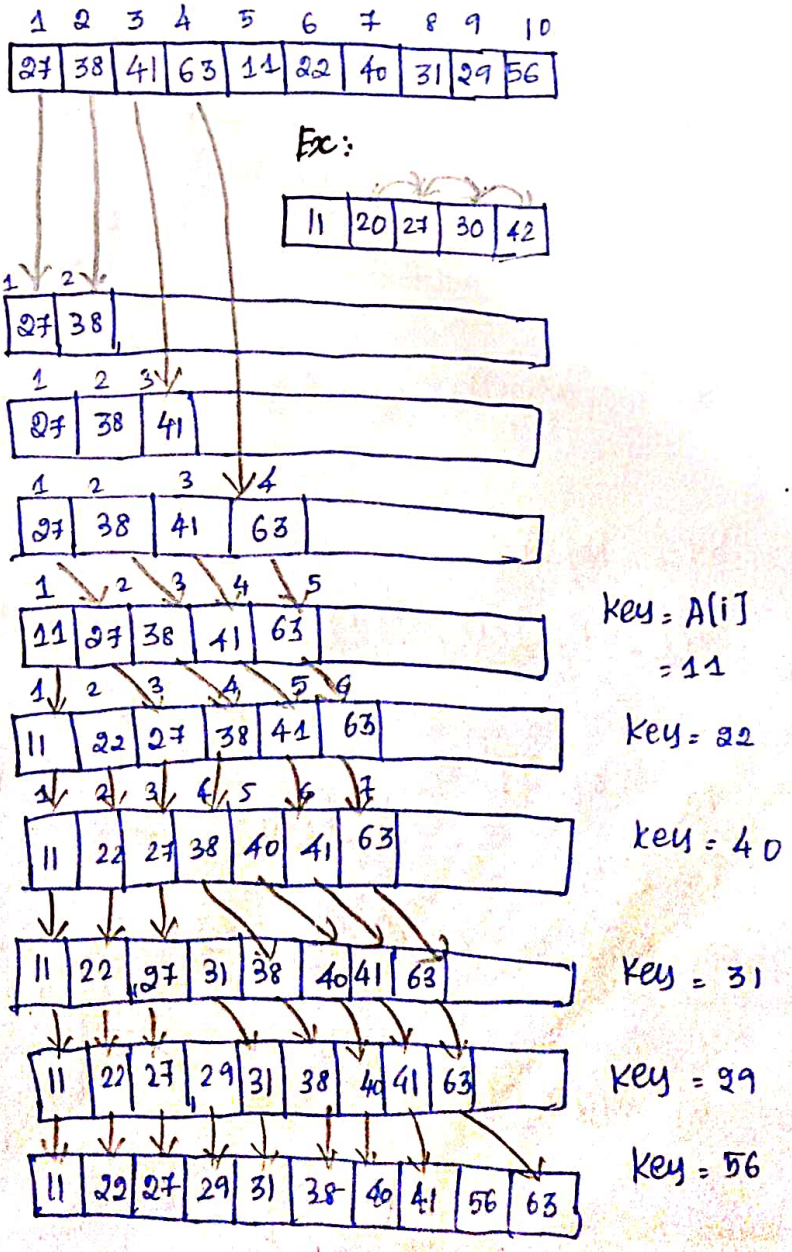
$\Theta(N * N * N)$

$\Theta(N^3)$

A $m \times n$ B $n \times p$

$N = \max(m,n,p)$

Insertion Sort:-



Algorithm INSERTION_SORT(A)

1. For j = 2 to A.length
2. Key = A[j]
3. i = j-1
4. while i > 0 and A[i] > Key
5. A[i+1] = A[i]
6. i = i+1
7. A[i+1] = Key.
8. end for.

Worst Case:

$t_j = j$

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$

$= 2+3+4+\cdots+n$

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

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

$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$

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

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

n	
n-1	
n	n-1
$\sum_{j=2}^{n} t_j$	
$\sum_{j=2}^{n} (t_j - 1)$	
$\sum_{j=2}^{n} (t_j - 1)$	
n-1	

$T(n) = O(n^2)$ in worst case

$T(n) = \Omega(n)$ in Best case

$T(n) = 4n - 3 + \frac{n^2+n-2}{2} + \frac{n^2-n}{2} + \frac{n^2-n}{2}$

$= \frac{3n^2}{2} + \frac{5}{2}n - 4$

```
#include <stdio.h>
int main()
{
  int a[50], n, i;
  void insertionSort (int [], int)
  Scanf ("%d", &n);
}
void insertionSort (int A[], int n)
{
  int j, Key, i;
  for (j=1; j<n; j++)
  {
    Key = A[j];
    i = j-1;
  while (i>=0 && A[i] >Key)
  {
    A[i+1] = A[i];
    i--;
```

```
for (i=0; i<n; i++)
  Scanf ("%d", &a[i]);
insertionSort (a,n);
for (i=0; i<n; i++)
  printf ("%d\t", a[i]);
```

Best case:

$t_j = 1$

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1$

$= n-1$

$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (1-1) = \sum_{j=2}^{n} 0 = 0$

$T(n) = n + n-1 + n-1 + n-1 + 0 + 1(n-1)$

$= 5n-4.$

$T(n) = \Omega(n)$

$T(n) = n + n-1 + n-1 + \sum_{j=2}^{n} t_j + \sum_{j=2}^{n} (t_j - 1) + \sum_{j=2}^{n} (t_j - 1) + n-1$

```
    }
    A[i+1] = key;
  }
}
```

PROGRAM:-

```
#include <stdio.h>
int main()
{
  int a[50], n, i;
  void insertion sort (int [], int)
  scanf ("%d", &n);
  for (i=0; i<n; i++)
  scanf ("%d", &a[i]);
  insertion sort (a,n);
  for (i=0; i<n; i++)
  printf ("%d\t", a[i]);
  }
  void insertion sort (int A[], int n)
  {
    int j, key, i;
    for (j=1; j<n; j++)
    {
      key = A[j];
      i = j-1;
    while (i>=0 && A[i] > key)
    {
      A[i+1] = A[i];
      i--;
    }
  } A[i+1] = key;
}
```

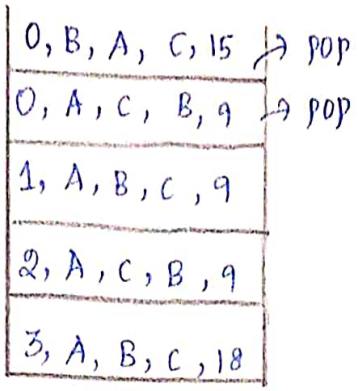
Tower of Hanoi:

```
typedef struct
{
  int N;
  char SN;
  char IN;
  char DN;
  int RA
} Element;
typedef struct {
      int length;
```

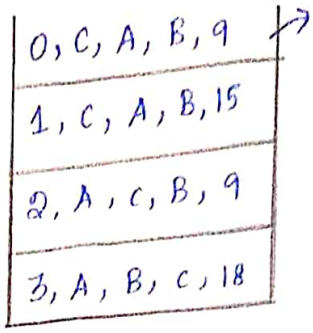
```
# include < stdio.h>
# include "Array stack.c"
# include <Math.h>
int main
   {
     int n, m=0;
     printf ("Enter number of disks:");
     scanf ("%d", &n);
     createstack (int pow(2, n));
     Element t = (n, 'A', 'B', 'C', 18};
     Step 2:
        push (t);
        t = peek ();
        if (t.N == 0)
              goto stop RA;
        else:
           {
           Element x = {t.N-1, t.SN, t.DN, t.IN, 9};
```

```
        t=x;
        goto step 2;
step 9:
        t= pop();
        t= peek();
        printf("\n move disc %d from %c to %c",t.N,
                                    t.SN, t.DN);
        t = peek();
        Element x2 = {t.N-1, t.SN, t.SN, t.DN, 15};
        t= x2
        goto step 2;
step 15:
        t= pop();
        t = peek();
        goto step RA;
Step RA:
        switch (t.RA) {
            case 9: goto step 9;
            case15: goto step 15;
            case 18: goto step 18;
        }
    step 18:
            printf("\n No. of moves = %d", m)",
            return 0;
    }
```

TRACING DIAGRAM:.

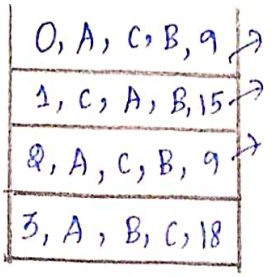


Move disc 1 from A to c

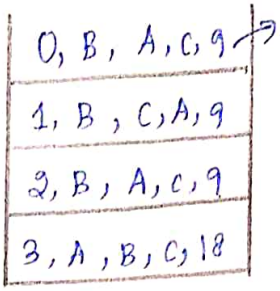


Move 2 from A to B

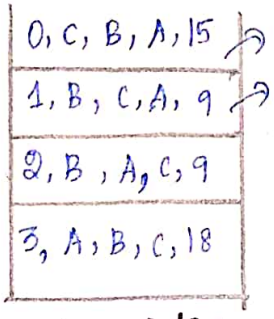
Move 1 from c to B



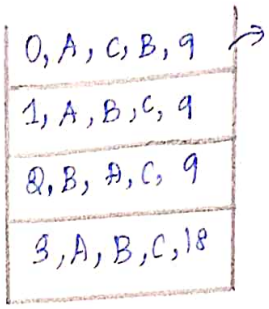
Move 3 from A to c



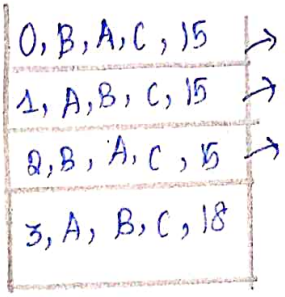
Move 1 from B to A



Move 2 from B to c



Move 1 from A to c



Tracing Tower of Hanoi:-

1. t = (3, A, B, C, 18)
2. PUSH (s, t)
3. t = PEEK (s) = (3, A, B, C, 18)
4. If t.N = 0 → 3 = 0 false
5. Else
7. t = (2, A, C, B, 9)
8. goto Step 2
2. PUSH (s, t)
3. t = PEEK (s) = (2, A, C, B, 9)
4. If t.N = 0 ⇒ 2 = 0 false
6. Else
7. t = (1, A, B, C, 9)
8. goto step 2
2. PUSH (s, t)
3. t = PEEK (s) = (1, A, B, C, 9)
4. If t.N = 0 ⇒ 1 = 0 false
6. Else
7. t = (0, A, C, B, 9)
8. goto step 2.
2. PUSH (s, t)
3. t = PEEK (s) = (0, A, C, B, 9)

4. If t.N = 0 ⇒ 0 = 0

5. goto step t.RA

9. t = POP(S)

10. t = PEEK (S) = (1, A, B, C, 9)

11. print

12. t = PEEK (S) = (1, A, B, C, 9)

13. t = (0, B, A, C, 15)

14. goto step 2.

2. PUSH (S, t)

3. t = PEEK(S) = (0, B, A, C, 15)

4. If t.N = 0 True

5. goto step 15

15. t = POP (S)

16. t = PEEK (S) = (0, B, A, C, 15)

17. goto step t.RA (9)

9. t = POP(S)

10. t = PEEK (2, A, C, B, 9)

11. print

12. t = PEEK(S) = (2, A, C, B, 9)

13. t = (1, C, A, B, 15)

14. goto step 2.

2. PUSH (S, t)

3. t = PEEK (S) = (1, C, A, B, 15)

4. If t.N = 0 ⇒ 1 = 0 false

5. Else

7. t = (0, C, B, A, 9)

8. goto step 2.

2. PUSH (s, t)

3. t = PEEK (s) = (0, C, B, A, 9)

4. If t.N = 0 True.

5. goto t.RA (9)

9. t = POP (s)

10. t = PEEK (s) = (1, C, A, B, 9)

11. print.

12. t = PEEK (s) = (1, C, A, B, 9)

13. t = (0, A, C, B, 15).

14. goto step 2

2. PUSH (s, t)

3. PEEK (s) = (0, A, C, B, 15)

4. If t.N = 0 ⇒ 0 = 0 True

5. goto step t.RA (15)

15. t = POP(s)

16. t = PEEK (s) = (1, C, A, B, 15)

17. goto step t.RA (15)

15. t = POP(s)

16. t = PEEK (s) = (2, A, C, B, 9)

17. goto step t.BA (9)

9. t = POP (S)

10. t = PEEK (S) = (3, A, B, C, 18)

11. print.

12. t = PEEK (S) = (3, A, B, C, 18)

13. t = (2, B, A, C, 15)

14. goto step 2.

2. PUSH (S, t)

3. t = PEEK (S) = (2, B, A, C, 15)

4. If t.N = 0 ⇒ 2 ≠ 0 false

6. Else

7. t = (1, B, C, A, 9)

8. goto Step 2

2. PUSH (S, t)

3. PEEK (S) = (1, B, C, A, 9)

4. If t.N = 0 (1 = 0) false

6. Else

7. t = (0, B, A, C, 9)

8. goto step 2.

2. PUSH (S, t).

3. PEEK (S) = (0, B, A, C, 9)

4. If t.N = 0 => (0 = 0) True

5. goto t.RA (9)

9. t = POP (S)

10. t = PEEK (S) = (1, B, C, A, 9)

11. print

12. t = PEEK (S) = (1, B, C, A, 9)

13. t = (0, C, B, A, 15)

14. goto step 2.

2. PUSH (S, t)

3. t = PEEK (S) = (0, C, B, A, 15)

4. If t.N = 0 => (0 = 0) True

5. goto t.RA

15. t = POP (S)

16. t = PEEK (S) = (1, B, C, A, 9)

17. goto step 9.

9. t = POP (S)

10. t = PEEK (S) = (2, B, A, C, 15)

11. print

12. t = PEEK (S) = (2, B, A, C, 15)

13. t = (1, A, B, C, 15)

14. goto step 2

8. PUSH (S, t)

3. t = PEEK (S).

3. PEEK (S) = (0, B, A, C, 9)

4. If t.N = 0 => (0 = 0) True

5. goto t.RA (9)

9. t = POP (S)

10. t = PEEK (S) = (1, B, C, D, 9)

11. print

12. t = PEEK (S) = (1, B, C, A, 9)

13. t = (0, C, B, A, 15)

14. goto step 2.

2. PUSH (S, t)

3. t = PEEK (S) = (0, C, B, A, 15)

4. If t.N = 0 => (0 = 0) True

5. goto t.RA

15. t = POP (S)

16. t = PEEK (S) = (1, B, C, A, 9)

17. goto step 9.

9. t = POP (S)

10. t = PEEK (S) = (2, B, A, C, 15)

11. print

12. t = PEEK (S) = (2, B, A, C, 15)

13. t = (1, A, B, C, 15)

14. goto step 2

2. PUSH (S, t)

3. t = PEEK (S).

4. If t.N=0 =)(1=0) false

6. Else

7. t=(0, A, C, B, 9)

8. goto step 2.

2. PUSH (S, t)

3. t= PEEK (S)

4. If t.N = 0 =) (0=0) True

5. goto step t.RA

9. t = POP (S)

10. t=PEEK(S)= (1, A, B, C, 9)

11. print

12. t=PEEK (S)= (1, A, B, C, 9)

13. t= (0, B, A, C, 15)

14. goto step 2

2. PUSH (S, t)

3. t=PEEK (S)

4. If t.N= 0 =) (0=0) True

5. goto t.RA (15)

15. t=POP(S)

16. t=PEEK (S) = (1, A, B, C, 15)

17. goto step t.RA

15. t= POP (S)

16. t=PEEK (S) = (2, A, B, C, 15)

17. goto step t.RA

15. t = POP(s)

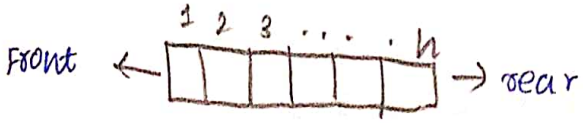
16. t = PEEK(s) = (3, A, B, C, 18)

17. goto t.RA

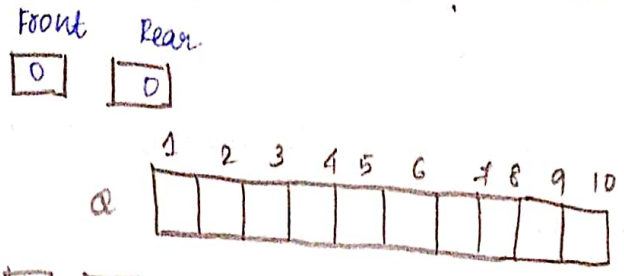
18. Return.

Queue :-

- FIFO (First In First Out)
- Insertion at rear end
- Deletion at front end



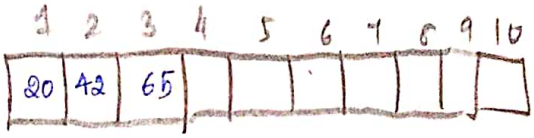
Empty Queue Font = Rear = 0



Insert 20

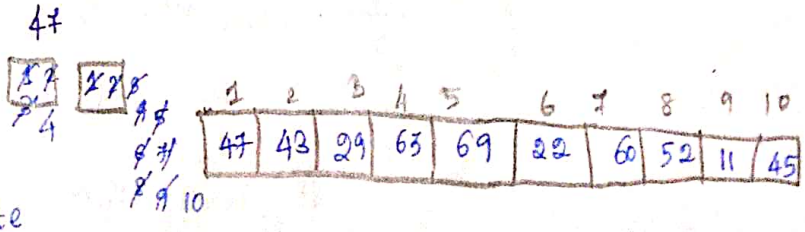
Insert 42

Insert 65



Insertion → ENQUEUE

Deletion → DEQUEUE



Delete

(20) (42) (65) (47) (43) (29)

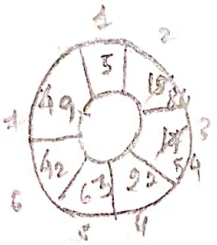
Algorithm ENQUEUE (Q, Front, Rear, X)

1. If Rear ≥ Q.length
2. print "Queue Overflows. Can't Insert"
3. Exit
4. Rear = Rear + 1
5. Q[Rear] = X
6. If Front = 0
7. Front = 1
8. Return

Algorithm DEQUEUE (Q, Front, Rear)

1. If Front = 0
2. print "Queue Empty. Can't delete"
3. Exit
4. X = Q[Front]
5. If Front = Rear
6. Front = Rear = 0
7. Else
8. Front = Front + 1
9. Return x

Circular Queue :-



5, 15, 17, 22, 63, 42, 49

Front 3

Rear ~~7~~ 2

[Rear % Q.length + 1]

If Front = Rear, the condition Overflows

Algorithm CENQUEUE (Q, Front, Rear, x).

1. If (Rear % Q.length + 1) = Front
2. Print "Queue Overflows . can't Insert"
3. Exit
4. Rear = (Rear % Q . length + 1)
5. Q [Rear] = x
6. If front = 0
7. Front = 1
8. Return

Algorithm CDENQUEUE (Q, Front, Rear).

1. If Front = 0
2. Print "Queue Empty . Can't delete"
3. Exit
4. X = Q [Front]
5. If Front = Rear
6. Front = Rear = 0
7. Else
8. Front = Front % Q. length + 1
9. Return x.

Size of Q : 5

1) Insert : 46, 2) Insert : 28, 3) Insert 64 4) Delete 5) Insert 72 6) Insert 11 7) Insert 24 8) Delete 9) Delete 10) Insert 49.

Q: 5 (size)

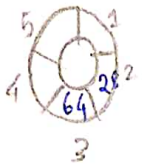
		Front	Rear
		0	0
1)	Insert : 46	1	1
2)	Insert : 28	1	2
3)	Insert : 64	1	3
4)	Delete	2	3
5)	Insert : 72	2	4
6)	Insert : 11	2	5
7)	Insert : 24	2	1
8)	Delete	3	1
9)	Delete	4	1
10)	Insert : 49	4	2

1)

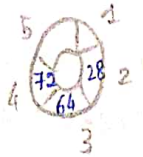
2)

3)

4)

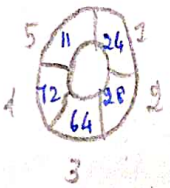


5)

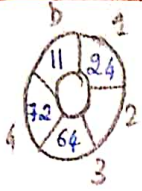


6)

7)



8)

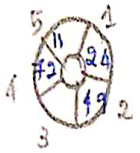


F = 3 R = 1

9)

F = 4 R = 1

10)

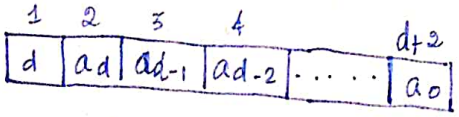


F = 4 R = 2

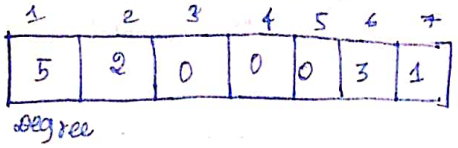
Polynomial :

$$P(d) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 \quad a_d \neq 0$$

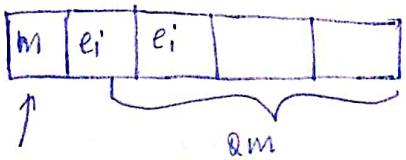
a_0 to a_{d-d-1}



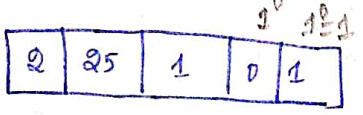
$P(5) = 2x^5 + 3x + 1$



$P(25) = x^{25} + 1$



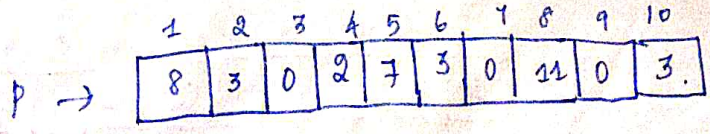
Non zero terms in the polynomial



$P = 3x^8 + 2x^6 + 7x^5 + 3x^4 + 11x^2 + 3$

$Q = 4x^2 + 3x + 7$

$P+Q = 3x^8 + 2x^6 + 7x^5 + 3x^4 + 15x^2 + 3x + 10.$



Q:

1	2	3	4
2	4	3	7

R →

1	2	3	4	5	6	7	8	9	10
8	3	0	2	7	3	0	15	3	10

Difference between degree = 8-2
= 6.

$P = 4x^5 - 3x^3 + 3x^2 + 7x + 5$

$Q = 11x^4 + 3x^3 + 4x^2 - 7x - 3$.

$P+Q = 4x^5 + 11x^4 + 7x^2 + 2$.

P →

1	2	3	4	5	6	7
5	4	0	-3	3	7	5

Q →

1	2	3	4	5	6
4	11	3	4	-7	-3

Difference between degrees = 5-4 = 1

R →

5	4	11	0	7	0	2

$P = 2x^{10} + 5x^2 + 3$

$Q = 3x^4 - 5x^2$

P →

1	2 (0x)	3 (0x)	4 (0x)	5	6 (0x)	7 (0x)
3	10	2	2	5	0	3

2M+1

Q →

1	2	3	4	5
2	4	3	2	-5

2N+1

R →

1	2	3	4	5	6
3	10	2	4	3	0	3

$P = 5x^{20} + 3x^7 + 4x^2 - 2$

$Q = 6x^{25} - 4x^8 - 3x^7 - 4x^2 + 2$

P →

1	2	3	4	5	6	7	8	9
4	20	5	7	3	2	4	0	-2

Q →

1	2	3	4	5	6	7	8	9	10	11
5	25	6	8	-4	7	-3	2	-4	0	0

R →

3	25	6	20	5	8	-4

Algorithm PADD (P[1.. 2m+1], Q [1.. 2n+1]

1. m = P[1]
2. n = Q[1]
3. Allocate R[1.. 2(m+n)+1]
4. i=2, j=2, k=2
5. While i ≤ 2m+1 and j ≤ 2n+1
6. if P[i] = Q[j] (exponent)
7. t = P[i+1] + Q[j+1]
8. if t ≠ 0.
9. R[k] = P[i]
10. R[k+1] = t
11. i = i+2, j = j+2, k = k+2.
12. Else if P[i] > Q[j]
13. R[k] = P[i]
14. R[k+1] = P[i+1]
15. i = i+2, k = k+2
16. Else
17. R[k] = Q[j]
18. R[k+1] = Q[j+1]
19. j = j+2, k = k+2
20. Endwhile
21. while j ≤ 2n+1
22. R[k] = Q[j]

23. R[k+1] = Q[j+1]

24. k = j+2 , k = k+2

25. While i ≤ 2m+1

26. R[k] = P[i]

27. R[k+1] = P[i+1]

28. i = i+2 , k = k+2

29. R[i] = k/2 / 1. (k-1)/2

30. Return R.

ALGORITHM SELECTION_SORT (A, n)

1. for i = 1 to n-1
2. minindex = i
3. for j = i+1 to n
4. if A[j] ≤ A[minindex]
5. minindex = j
6. if minindex ≠ i
7. temp = A[i]
8. A[i] = A[minindex]
9. A[minindex] = temp.

n
n-1
n+(n-1)+...+1 = n(n+1)/2
n(n-1)/2
0 to n(n-1)/2
0 to n-1
0 to n-1
0 to n-1

Worst Case Analysis:-

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

$$= 6n - 5 + 3\frac{n^2}{2} - \frac{n}{2} = \frac{3n^2}{2} - \frac{11n}{2} - 5$$

$$= O(n^2)$$

Algorithm RECURSIVE FACTORIAL (n)

1. if n = 0
2. return 1
3. else
4. return n * RFACT (n-1)

RFACT (4)

1. if n = 0 [4 = 0] false]
3. else
4. return (4 * RFACT (3)) 24 (6)

1. if n = 0 [3 = 0 False]
3. else
4. return (3 * RFACT(2)) 6 (2)

1. if n = 0 [2 = 0 False]
3. else
4. return (2 * RFACT(1)) 2 1

1. if n = 0 [1 = 0 False]
3. else
4. return (1 * (RFACT(0)) 1 1

1. if n = 0 [0 = 0 True]
2. return 1

4! = 24

Algorithm RSUM (A, n)

1. if ~~n=1~~
2. return A[1]
3. else
4. return A[n] + RSUM [A, n-1]

	1	2	3	4	5
A	20	11	65	30	15

$Sum(A, n) = A[n] + sum(A, n-1)$ if $n > 1$

$sum(A, 1) = A[1]$

RSUM (A, 5)

1. if n = 1 (1 = 5) false
3. else
4. return A[5] (15) + RSUM (A, 4) (126)

    1. if n = 1 (4 = 4) false
    3. else
    4. return A[4] (30) + RSUM (A, 3) (96)

        1. if n = 1 (3 = 3) false
        3. Else
        4. return A[3] (65) + RSUM (A, 2) (31)

            1. if n = 1 (2 = 2) false
            3. Else
            4. return A[2] (11) + RSUM (A, 1) (20)

                1. if n = 1
                2. return A[1] (20).

141 → 20 + 31 + 96 + 126

Algorithm RFIBONACCI (n)

1. if n = 1
2. return 0
3. else if n = 2
4. return RFIB (n-1) + R (FIB) (n-2)

RFIB (4) = 2.

1. If n = 1 [4 = 1 false]
3. else if n = 2 [4 = 2 false]
5. else
6. return RFIB(3) + RFIB(2)

1. if n = 1 [3 = 1 False]
3. else if n = 2 [3 = 2 false]
5. else
6. return RFIB(2) + RFIB(1)

1. false
3. True
4. return 1.

1. If n = 1 (1 = 1)
2. return 0.

1. if n = 2 [2 = 1 false]
3. else if n = 2 [2 = 2 True]
4. return 1

1. if n = 1 [1 = 1 True]
2. return 0.

Algorithm RLSEARCH (A, n, x)

1. if n = 0
2. return -1.
3. else if A[n] = x
4. return n
5. else
6. return RLSEARCH (A, n-1, x)

	1	2	3	4	5
A	20	11	65	30	15

Trace for x = 11

Trace for x = 32

RLSEARCH (A,5,11)

1. if n=0 [5=0 False]
2. else if A[5]=11
6. return RLSEARCH (A,4,11)

1. if n=0 [4=0 FALSE]
3. else if A[4]=11
6. return RLSEARCH (A,3,11)

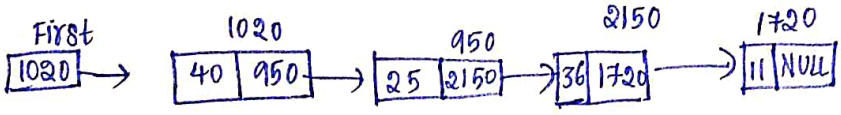
1. if n=0 [3=0 False]
3. else if A[3]=11
6. return RLSEARCH (A,2,11)

1. if n=0 [2=0] false
3. else if A(2)=11
4. return 2.

LINKED LIST:

T-data
T-Link

40, 25, 36, 11



Self referential structure:

```
struct list
{
    int data;
    struct list * link;
};
```

```
struct list *t;
Allocation:
t = (struct list *) malloc (size of (struct list));
Deallocation:
free (t); ⟶ FREENODE(t)
```

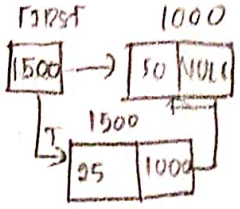
Algorithm
t = GETNODE(),
return a memory allocation

t = FREENODE(t)
Algorithm for deallocation.

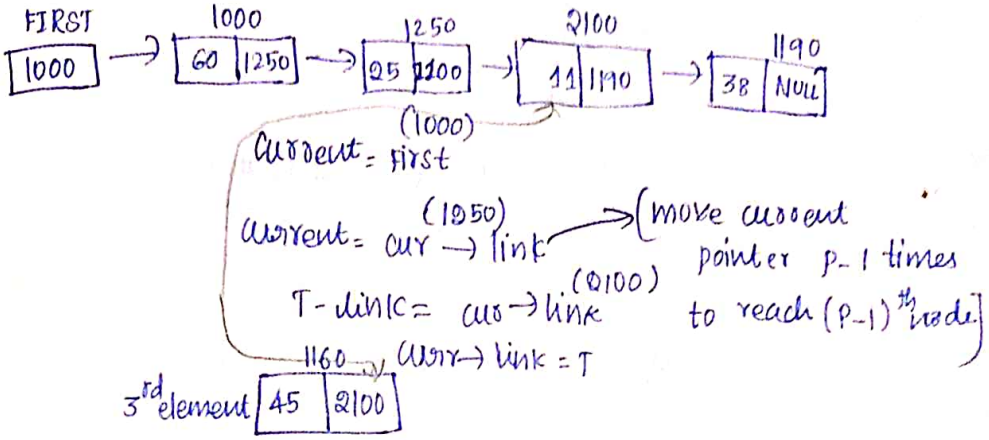
INSERTION - AT - BEGINING Algorithm:

Alg INSERT_AT-BEG(FIRST, x)

1. t = GETNODE()
2. t → data = x
3. t → link = FIRST
4. FIRST = t



T = data = x T → data = x
T → link = NULL T → link = FIRST
FIRST = T FIRST = T



INSERTION - AT BEG

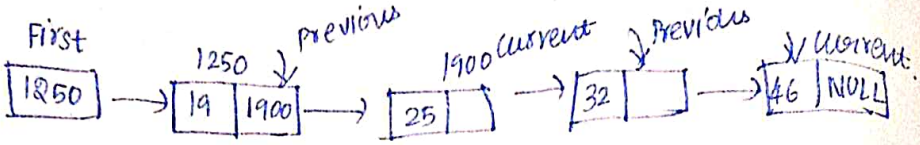
Algorithm INSERT - AT - LOCATION (FIRST, x, p) ← position

1. t = GETNODE()
2. t → data = x
3. t → link = NULL
4. if FIRST = NULL or p = 1
5. t → link = FIRST
6. FIRST = t
7. return
8. cur = FIRST
9. count = 1
10. While cur → link ≠ NULL and count < p-1

11. cur = cur → link
12. count = count + 1
13. t → link = cur → link
14. cur → link = t
15. return

Previous = NULL
cur = FIRST (1250)

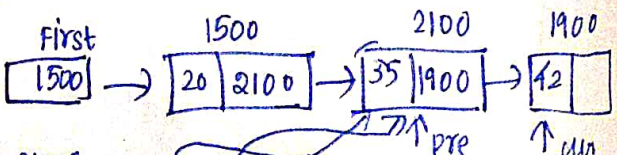
previous = current (1250)
current = cur → link (1900)



Algorithm INSERT_AT_ORD (FIRST, x)

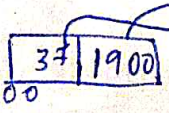
1. T = GETNODE ()
2. T → data = x
3. T → link = NULL
4. if FIRST = NULL (or) first → data > x.
5. T → link = FIRST
6. FIRST = T
7. Else
8. prev = NULL
9. cur = FIRST
10. while cur != NULL and cur → data < T → data
11. prev = cur
12. cur = cur → link
13. if prev = NULL
14. FIRST = T
15. else
16. ← T → link = cur
17. ← prev → link = T
18. return.

Ex:



pre = NULL → 2100 → 1900

cur → 1500 → 2100 → 1900



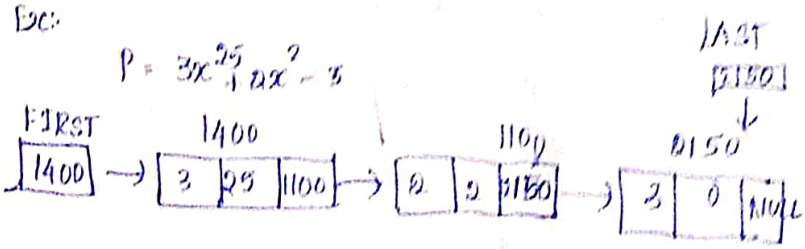
20 < 37
35 < 37
42 ≮ 37

Polynomial Representation:

- Node structure

Ex:

$P = 3x^{25} + 2x^2 - 5$



Creation of polynomial:

```
Algorithm Create poly ()
// Create a polynomial P
1. P.First = P.Last = NULL
2. While there is input
3.     Read coef, exp
4.     INSERT-ORD (P, coef, exp)
5. Return P.
```

INSERTING A TERM IN ORDER:

```
Algorithm INSERT_ORD (P, coef, exp)
// To insert a new term of the polynomial in the
// decreasing order its exponent.
1. n = GETNODE ()
2. n → coef = coef
3. n → exp = exp
4. n → link = NULL
5. If P.First = NULL
6.     P.First = P.Last = n
7. else
8.     prev = NULL
9.     cur = First
10.    while cur != NULL and cur → exp >
                                n → exp
```

```
11.     prev = cur
12.     cur = cur → link
13. if prev = NULL
14.     n → link = P.First
15.     P.First = n
16. Else
17.     n → link = cur
18.     prev → link = n
19.     if cur = NULL
20.     P.last = n
21. Return P.
```

INSERTING A TERM A END:

Algorithm INSERT_AT_LAST(P, coef, exp)

// To insert a new term of the polynomial at end.

```
1. n = GETNODE()
2. n → coef = coef
3. n → exp = exp
4. n → link = NULL
5. if P.First = NULL
6.     n → link = P.First
7.     P.First = P.Last = n
8. Else
9.     n → link = P.last → link
10.    P.Last → link = n
11.    P.Last = n
12. Return P.
```

DISPLAYING THE POLYNOMIAL:

Algorithm Displaypoly (P)

// To display the polynomial

1. If P.First = NULL
2. Print 'Empty polynomial'
3. Else
4. Temp = P.First
5. While Temp → link != NULL
6. Print Temp → coef, "x^", Temp → exp, "+"
7. Temp = Temp → link
8. Print Temp → coef, "x^", Temp → exp.
9. Return.

ADDING TWO POLYNOMIALS

Algorithm ADD_POLY (P, Q)

// Adding two polynomials P and Q.

1. R.First = R.Last = NULL
2. t1 = P.First
3. t2 = Q.First
4. While t1 != NULL and t2 != NULL
5. If t1 → exp > t2 → exp
6. R = INSERT_AT_LAST(R, t1 → coef, t1 → exp)
7. t1 = t1 → link
8. Else if t1 → exp < t2 → exp
9. R = INSERT_AT_LAST (R, t2 → coef, t2 → exp)
10. t2 = t2 → link
11. Else
12. coef = t1 → coef + t2 → coef
13. exp = t1 → exp.
14. if coef != 0
15. R = INSERT_AT_LAST (R, coef, t2 → exp)

16. t1 = t1 → link

17. t2 = t2 → link

18. While t1 != NULL

19. R = INSERT_AT_LAST(R, t1 → coef, t1 → exp)

20. t1 = t1 → link

21. While t2 != NULL

22. R = INSERT_AT_LAST(R, t2 → coef, t2 → exp)

23. t2 = t2 → link

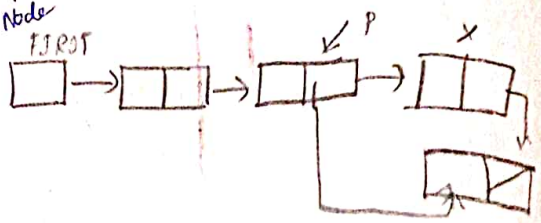
24. Return R.

DELETION :.

Deleting a node in SLL which comes after node P.

Algorithm DELETE NODE (FIRST, P)

1. X = P → link (Address of Before Node)
2. P → link = X → link
3. RETNODE (x)
4. Return

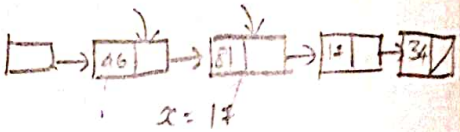


Deletion :

Deleting a node in SLL which has value x.

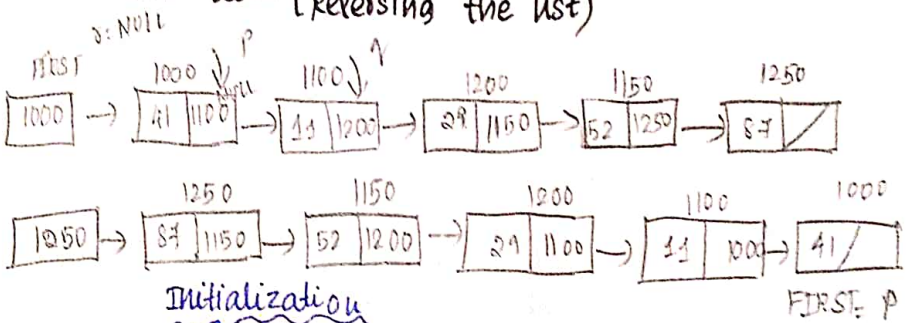
Algorithm DELETE_SLL (FIRST, x)

1. If FIRST = NULL
2. Print "List Empty. Can't delete".
3. Return
4. If FIRST → data = x
5. T = FIRST
6. FIRST = FIRST → link
7. Else



8. Prev = FIRST
9. cur = FIRST → link
10. While cur != NULL and cur → data ≠ x
11. prev = cur
12. cur = cur → link
13. if cur = NULL
14. Print " Element is not present in the list. can't delete"
15. Return
16. prev → link = cur → link
17. X = cur
18. RETNODE (T)

Inserting the list (Reversing the list)



Initialization

r = NULL

P = FIRST

q = P → link

change

P → link = r

Moving to next node

r = p

P = q

q = q → link

repeated while P ≠ NULL

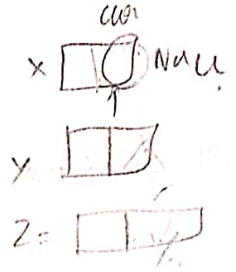
Algorithm INSERT (FIRST) :-

1. If FIRST = NULL
2. Return
3. r = NULL

4. P = FIRST
5. q = P → link
6. While q ≠ NULL
7. P → link = r
8. r = p
9. p = q
10. q = q → link
11. FIRST = p
12. Return FIRST

Join
Concatenation of two lists x and y:-

1. If X = NULL
2. Z = Y
3. else if Y = NULL
4. Z = X
5. else
6. cur = x
7. While cur → link != NULL
8. cur = cur → link
9. cur → link = y
10. Z = X
11. Return z.



Algorithm INSERT_AT_BEG (HEAD, x)

1. T = GETNODE ()
2. T → data = x
3. T → link = HEAD → link
4. HEAD → link = T
5. HEAD → data = HEAD → data + 1.
6. Return

Algorithm INSERT_AT_LOC (HEAD, x, p).

1. T = GETNODE ()
2. T → data = x
3. T → link = NULL
4. cur = HEAD
5. count = 1
6. while cur → link ≠ NULL and count ≤ p - 1.
7. cur = cur → link
8. count = count + 1
9. T → link = cur → link
10. cur → link = T
11. HEAD → data = HEAD → data + 1
12. Return.

Node which stores the Head data → length of the linked list

Algorithm LENGTH (HEAD)

1. Return HEAD → data

Algorithm LENGTH (FIRST)

1. Count = 0
2. Cur = FIRST
3. While cur ≠ NULL
4. Count = count + 1
5. cur = cur → link
6. Return Count.

Deleting a node in singly linked list with HEAD having element x.

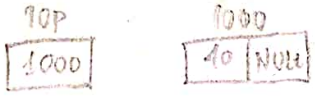
Algorithm DELETE_SLL (HEAD, x)

1. Cur = HEAD
2. While cur → link → data ≠ x
3. Cur = Cur → link
4. if cur → link = NULL. 5. print "Element not present. Can't delete"
7. T = cur → link 6. Return.
8. cur → link = T → link
9. HEAD → data = HEAD → data - 1
10. RETNODE (T)
11. Return.

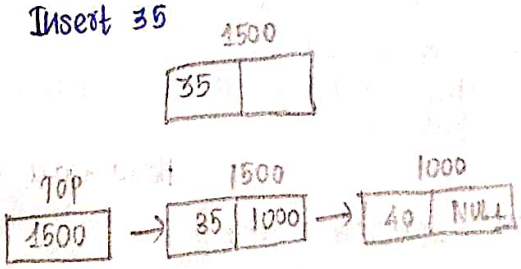
Linked stack: → (LIFO) Last In First Out

Top is ptr with points the element entered lately.

TOP = NULL → Empty stack.



Insert 35



Algorithm PUSH (S.TOP, x):-

1. T = GETNODE ()
2. T→data = x
3. T→link = TOP
4. TOP = T
5. Return

Algorithm POP (S.TOP):-

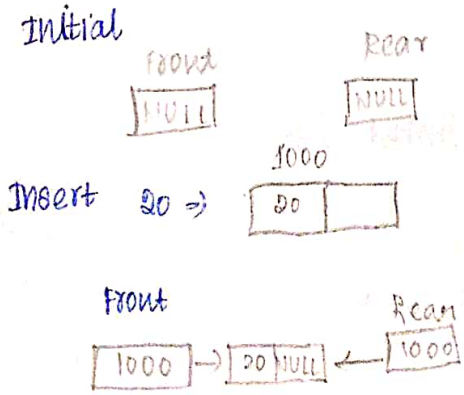
1. If TOP = NULL
2. print "stack Empty . can't delete"
3. Return
4. X = TOP→data
5. T = TOP
6. TOP = T→link
7. RETNODE(T)
8. Return X.

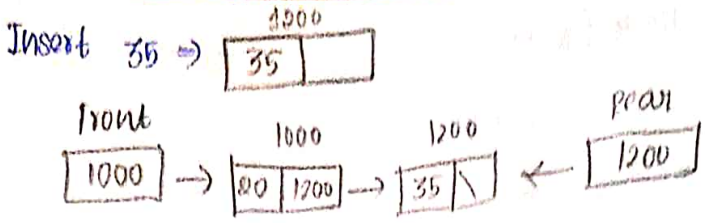
Algorithm PEEK (S.TOP):-

1. If Top = NULL
2. print "stack Empty. No peek"
3. Return
4. x = Top→data
5. Return x.

Linked Queue:→ FIFO, front, Rear.

ptr





Algorithm ENQUEUE (Q, Front, Rear, x):-

1. T = GETNODE ()
2. T→data = x
3. T→link = NULL
4. If REAR = NULL
5. FRONT = REAR = T
6. Else
7. REAR→link = T
8. REAR = T
9. Return

Algorithm DEQUEUE (Q, Front, Rear):-

1. If FRONT = NULL
2. print "Queue Underflow. Can't delete"
3. Return
4. X = FRONT→data
5. T = FRONT
6. If FRONT = REAR
7. FRONT = REAR = NULL.
8. Else.
9. FRONT = FRONT→link.
10. RETNODE (T).
11. Return X.

Circular Singly Linked List:-

Here the last node's link field is pointing the first node.

Algorithm INSERT_AT_BEG (circular singly linked list) CALL (FIRST, x) ::

1. T = GETNODE ( )
2. T→data = x
3. If FIRST = NULL
4. T→link = FIRST = T
5. Return
6. T→link = FIRST
7. aux = FIRST.
8. while aux→link ≠ FIRST
9. aux = aux→link
10. aux→link = T.
11. FIRST = T.

Deletion when (x) is given and prev add of the node to be deleted.

Doubly Linked List:


```
Struct dllnode
{
    int data;
    Struct dllnode *prev ;  (Predecessor, left link)
    Struct dllnode *next ;  (Successor, Right link)
};
```

Initially,

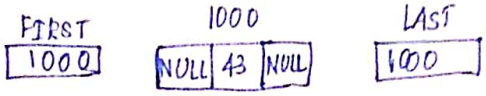
FIRST = NULL

LAST = NULL

43, 20, 65, 11

↓

Node Name = T



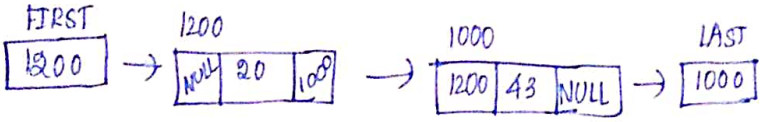
T = GETNODE ()

T → data = x

T → prev = NULL

T → next = NULL

FIRST = LAST = T



T = GETNODE ()

T → data = x

T → prev = NULL

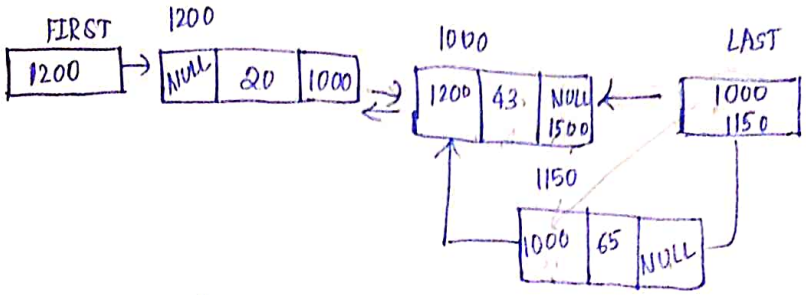
T → Next = FIRST

FIRST → prev = T

FIRST = T.

Algorithm INSERT_BEG_DLL (FIRST, LAST, x).

1. T = GETNODE()
2. T→data = x
3. T→prev = NULL
4. T→next = FIRST
5. If FIRST = NULL
6. FIRST = LAST = T
7. Else
8. FIRST→prev = T
9. FIRST = T



T = GETNODE ()

T→data = x

T→next = NULL

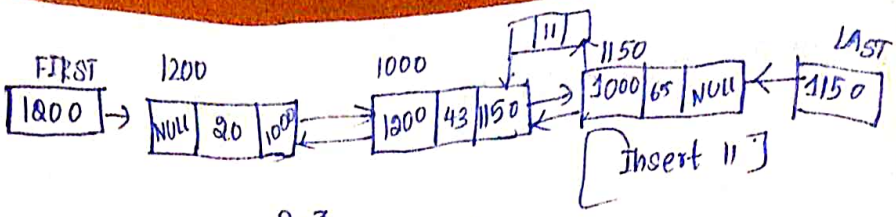
T→prev = LAST

LAST→next = T

LAST = T.

Algorithm INSERT_AT_END_DLL (FIRST, LAST, x).

1. T = GETNODE ()
2. T→data = x
3. T→prev = LAST
4. T→next = NULL
5. If FIRST = NULL
6. FIRST = LAST = T
7. Else
8. LAST→next = T
9. LAST = T.



P = 3

Moving upto $P-1^{th}$ node.

cur = ~~1200~~ ~~1000~~ 1150

count = ~~1~~ ~~2~~ 3

```
cur = FIRST
count = 1
While cur ≠ NULL and count < P
    cur = cur → next
    count = count + 1
```

After inserting 11,

```
T = GETNODE()
T→data = x
T→prev = cur → prev
T→next = cur
cur → prev → next = T
cur → prev = T.
```

Algorithm INSERTAT_LOC DLL (FIRST, LAST, x).

1. T = GETNODE ()
2. T→data = x
3. T→prev = NULL
4. T→next = NULL
5. If FIRST = NULL
6. FIRST = LAST = T
7. Else
8. cur = FIRST
9. count = 1
10. While cur ≠ NULL and count < p.
11. cur = cur → next
12. count = count + 1
13. If cur = NULL ;
14. T→prev = LAST
15. LAST→next = T
16. LAST = T.

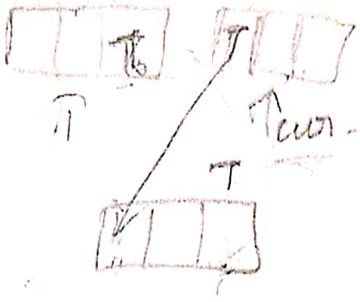
17. Else

18. T→prev = cur→prev

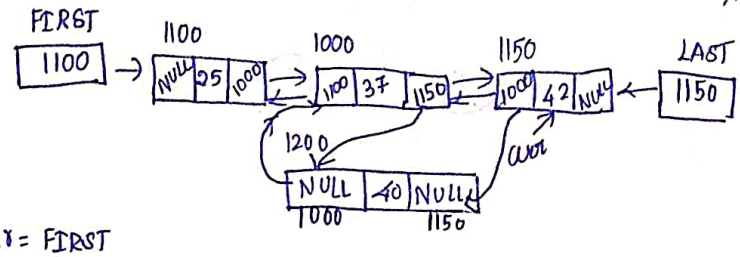
19. T→next = cur

20. cur→prev→next = T

21. cur→prev = T.



Algorithm INSERT_AT_ORD DLL (FIRST, LAST, x, p).



cur = FIRST

while cur ≠ NULL and

cur→data < x

cur = cur→next

if cur ≠ NULL

T→prev = cur→prev

T→next = cur

cur→prev→next = T

cur→prev = T.

T = GETNODE()

T→data = x

T→prev = NULL

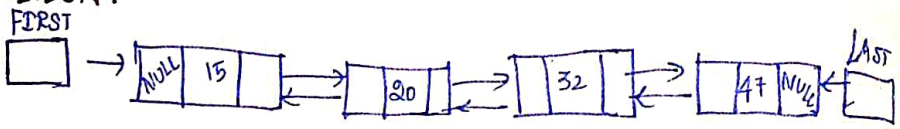
T→next = NULL

Algorithm INSERT_DLL_ORD (FIRST, LAST, x)

1. T = GETNODE()
2. T→data = x
3. T→prev = NULL
4. T→next = NULL
5. If FIRST = NULL
6. FIRST = LAST = T
7. return
8. If FIRST→data > x
9. T→next = FIRST
10. FIRST→prev = T
11. FIRST = T
12. return.

13. cur = FIRST
14. While cur ≠ NULL and cur → data < x
15. cur = cur → next] True condition.
16. If cur ≠ NULL
17. [T → prev = cur → prev
18. T → next = cur
19. cur → prev → next = T
20. cur → prev = T
21. else]
22. T → prev = LAST
23. LAST → next = T
24. LAST = T
25. return.

DELETION :



cur → prev → next = cur → next

cur → next → prev = cur → prev

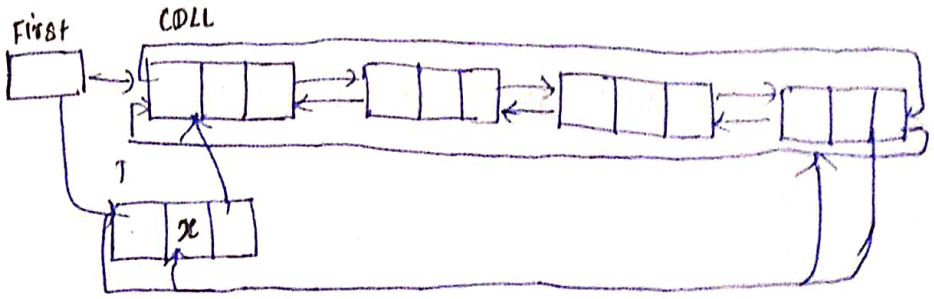
Algorithm DELETE_DLL (FIRST, LAST, x).

1. If FIRST = NULL
2. print "List Empty . Can't delete"
3. Return
4. cur = FIRST
5. While cur ≠ NULL and cur → data ≠ x
6. cur = cur → next
7. If cur = NULL
8. print " Element", x, "is not present in DLL"
9. Else
10. if cur → prev = NULL
11. FIRST = T
12. Else
13. cur → prev → next = cur → next

and if

14. If cur→next = NULL
15. LAST = cur → prev
16. Else
17. end if cur → next → prev = cur → prev
18. RETNODE (cur).
end if (10)

b) INSERTION_AT_BEG (FIRST, x)



T→ next = FIRST

T→ prev = FIRST → prev

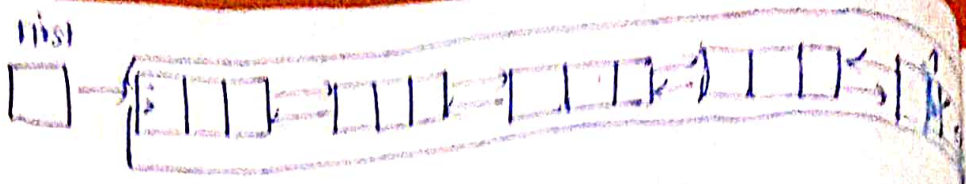
FIRST → prev → next = T

FIRST → prev = T

FIRST = T

Algorithm INSERT_AT_BEG (FIRST, x)

1. T = GETNODE()
2. T→ data = x
3. If FIRST = NULL
4. T→ prev = T
5. T→ next = T
6. FIRST = T
7. else
8. T→ next = FIRST
9. T→ prev = FIRST→ prev
10. FIRST → prev → next = T
11. FIRST→ prev = T
12. FIRST = T



T→prev = FIRST→prev

T→next = FIRST

FIRST→prev→next = T

FIRST→prev = T.

Algorithm INSERT_AT_END_CDLL (FIRST, x).

1. T = GETNODE()
2. T→data = x
3. If FIRST = NULL
4. T→prev = T
5. T→next = T
6. FIRST = T
7. else
8. T→prev = FIRST→prev
9. T→next = FIRST
10. FIRST→prev→next = T
11. FIRST→prev = T.

Algorithm INSERT AT_LOC CDLL (FIRST, x, p).

1. T = GETNODE()
2. T→data = x.
3. If FIRST = NULL
4. T→prev = T
5. T→next = T
6. FIRST = T
7. return
8. If P = 1
9. T→prev = FIRST→prev
10. T→next = FIRST
11. FIRST→prev→next = T
12. FIRST→prev = T
13. FIRST = T

14. return
15. cur = FIRST
16. count = 1
17. While cur → next ≠ FIRST and count < p-1.
18. cur = cur → next
19. count = count + 1
20. T → prev = cur
21. T → next = cur → next
22. cur → next → prev = T
23. cur → next = T
24. return.

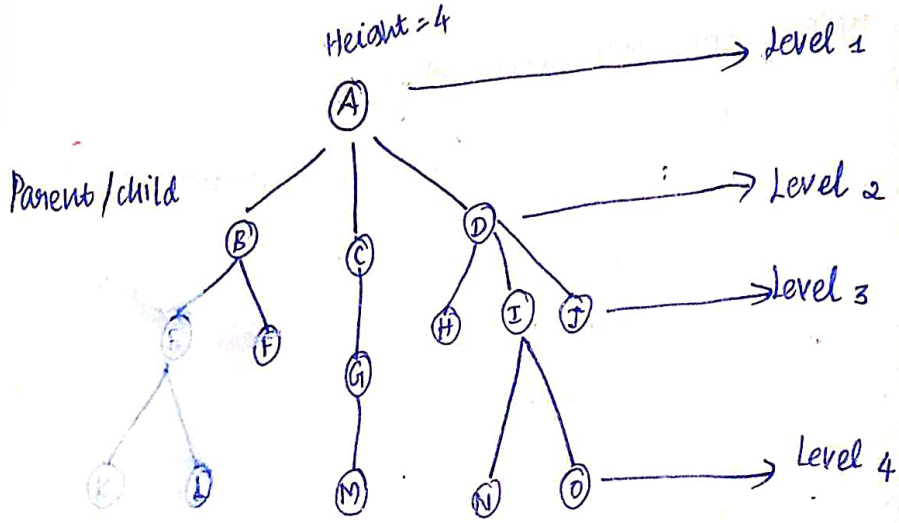
INSERT_ORD_CDLL (FIRST, x)

1. T = GETNODE()
2. T → data = x
3. If FIRST = NULL
4. T → prev = T
5. T → next = T
6. FIRST = T
7. return
8. If FIRST → data > x
9. T → prev = FIRST → prev
10. T → next = FIRST
11. FIRST → prev → next = T
12. FIRST → prev = T
13. FIRST = T
14. Return.
15. cur = FIRST
16. While cur → data < x and cur → next ≠ FIRST
17. cur = cur → next
18. T → prev = cur
19. T → next = cur → next
20. cur → next → prev = T
21. cur → next = T
22. Return.

Non-Linear Data Structures

Trees

- Non linear data structures.
- finite set of nodes.
- One specially designated node called as root.
- Each node in the tree has zero or more subtrees.
- A node with no subtree is called as leaf node.



→ Node with no child is called leaf node or terminal node.

Eg: F, K, L, M, H, J, N, O

→ Other nodes are non terminal nodes or internal nodes.

→ Root has no parent.

→ Number of children, the node has, is known as Degree

→ Degree of tree is known as maximum number of any node in that tree.

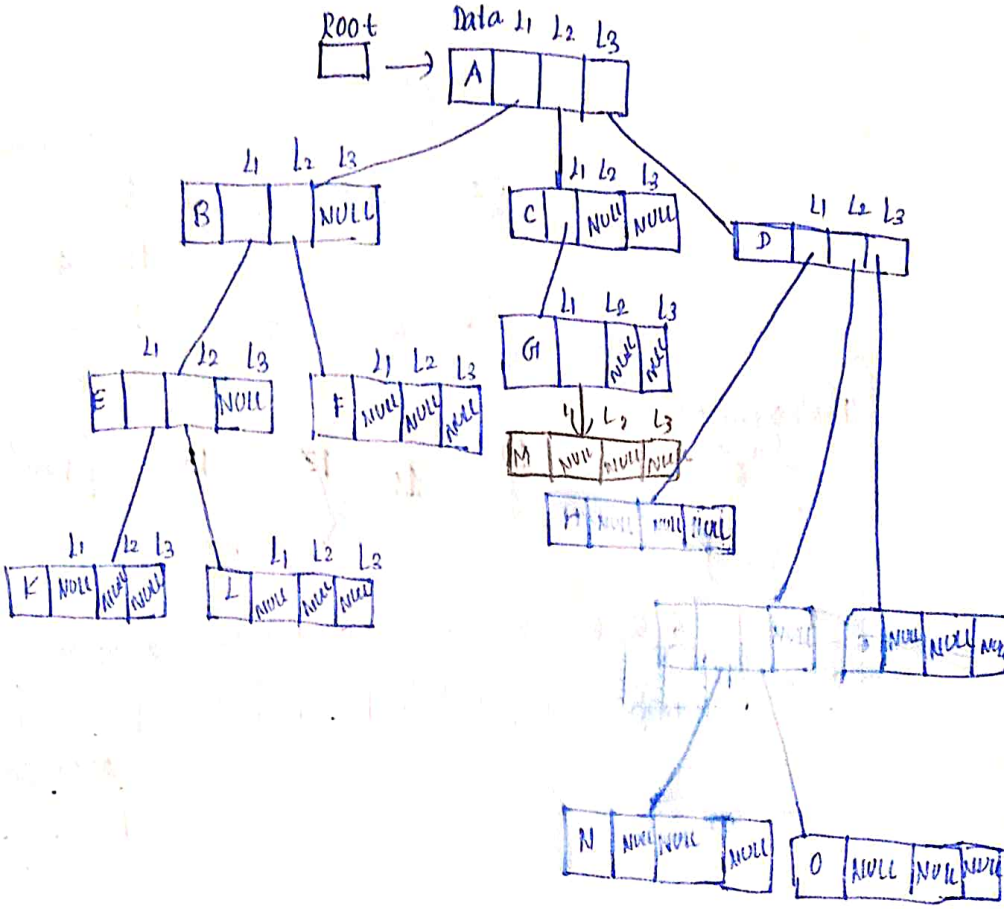
→ children of the same parent is called sibling.

→ Ancestor / Descendant.

1) Paranthetical Representation:-

(A, (B(E(K,L), F), C(G(M)), D(H, I(N,O), J)))

2) Linked Representation:-

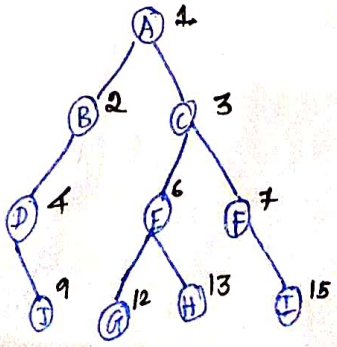


Binary tree:-

Degree of tree = 2.

① Sequential Representation

② Linked representation.



Sequential Representation:

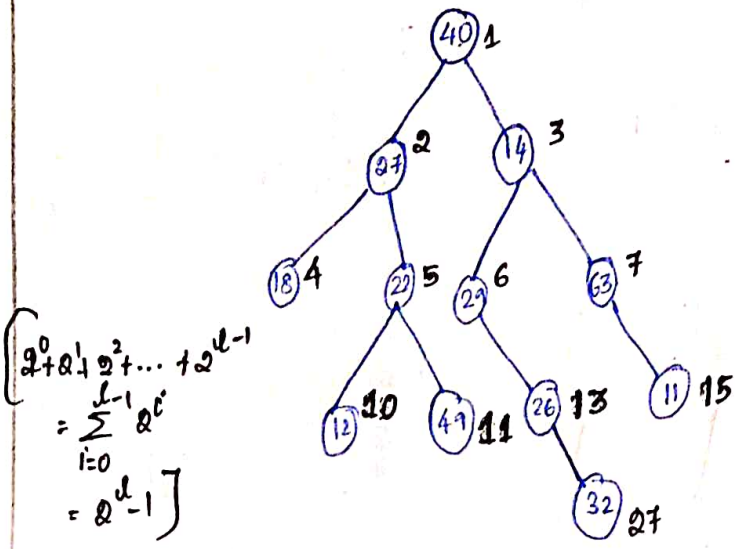
Root node - 1

node - i

left child - 2*i

Right child - 2*i+1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	-	E	F	-	J	-	-	G	H	-	I



Maximum nodes

l1 - 1 2^0

l2 - 2 2^1

l3 - 4 2^2

l4 - 8 2^3

$[l \to 2^{l-1}]$

$2^5 - 1$

$= 32 - 1$

$= 31$

$[2^0 + 2^1 + 2^2 + \dots + 2^{l-1}$

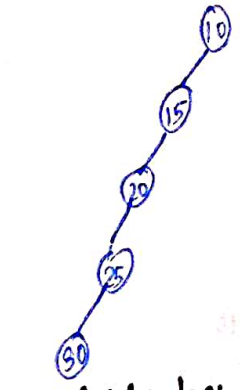
$= \sum_{i=0}^{l-1} 2^i$

$= 2^l - 1]$

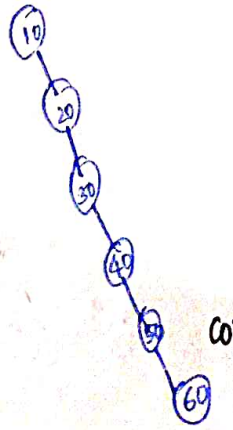
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
40	27	14	18	22	29	63	-	-	12	49	-	26	-	11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Skewed Tree:

the tree has only left or right child is known as skewed Tree.



completely left skewed tree

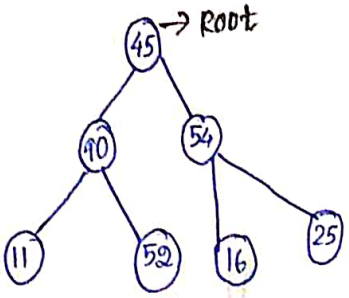


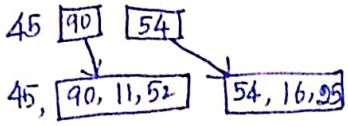
completely Right skewed tree

Preorder - Data Left Right (DLR)

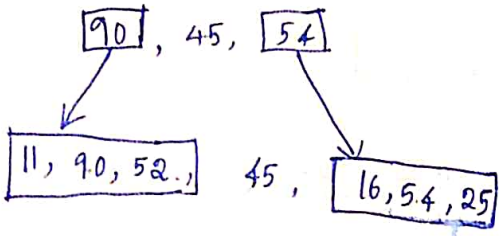
Inorder - Left Data Right (LDR)

Postorder - Left Right Data (LRD)

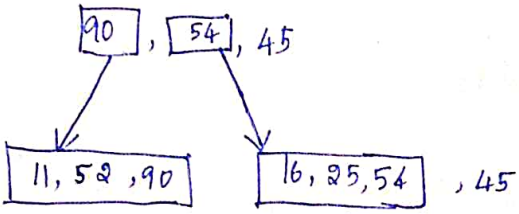




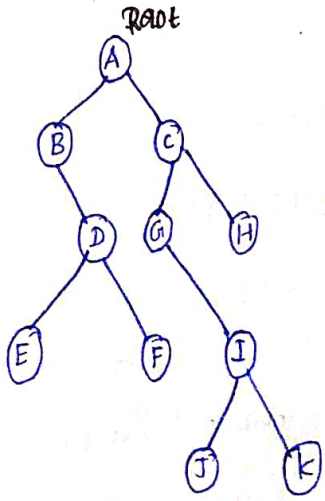
Preorder : 45, 90, 11, 52, 54, 16, 25.



Inorder : 11, 90, 52, 45, 16, 54, 25.



Postorder : 11, 52, 90, 16, 25, 54, 45.



Preorder:- D, L, R

A [B] [C]

A B [D] C [G] H

A B D E F C G [I] H

Preorder:- A, B, D, E, F, C, G, I, J, K, H.

Inorder:- L, D, R.

[B] A [C]

B [D] A [G] C H

B E D F A G [I] C H

Inorder:- B, E, D, F, A, G, J, I, K, C, H

Postorder:- L, R, D.

[B] [C] A

[D] B [G] H C A

[E] F D B [I] G H C A

E F D B J K I G H C A

Postorder:- E, F, D, B, J, K, I, G, H, C, A

Algorithm PREORDER (T):-

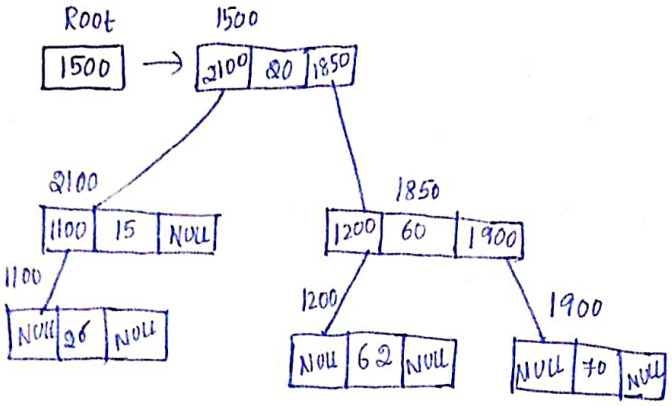
1. If $T \neq NULL$
2. print T→data
3. PREORDER (T→lchild)
4. PREORDER (T→rchild)

Algorithm INORDER (T):-

1. If $T \neq NULL$
2. INORDER (T→lchild)
3. print T→data
4. INORDER (T→rchild)

Algorithm POSTORDER (T):.

1. If T≠NULL
2. POSTORDER (T→ lchild)
3. POSTORDER (T→ rchild)
4. Print T→ data.



PREORDER (1500) — T

1. T≠ NULL True
2. print T→ data (80)
3. PREORDER (2100) — T
   1. T≠NULL True
   2. print T→ data (15)
   3. PREORDER (1100) — T
      1. If T≠NULL True
      2. print T→ data (26)
      3. PREORDER (NULL) — T
         1. T≠ NULL False
      4. PREORDER (NULL)
         1. T≠ NULL False
   4. PREORDER (NULL)
      1. T≠ NULL False
4. PREORDER (1850)
   1. T≠ NULL True
   2. print T→ data (60)
   3. PREORDER (1200)

1. T≠NULL True
2. print T→data (62)
3. PREORDER (NULL)
   1. T≠NULL False
4. PREORDER (NULL)
   1. T≠ NULL False

4. PREORDER (1900)

1. If T ≠ NULL (True)
2. Print T→data (70)
3. PREORDER (NULL)
   1. If T≠ NULL (False)
4. PREORDER (NULL)
   1. If T≠ NULL (False)

Inorder:. 24 46 75

11 24 39 46 62 75 82

7 11 24 39 40 46 62 63 75 82 92

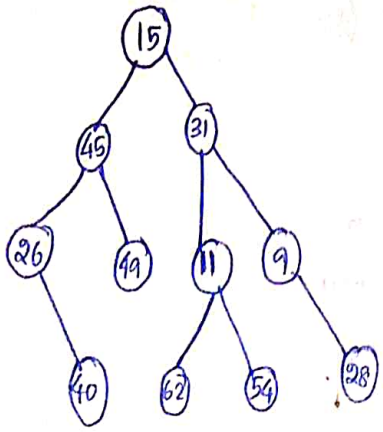
7 11 24 39 40 46 62 63 74 75 82 89 92.

Postorder : 24 75 46

11 39 24 62 82 75 46

7 11 40 39 24 63 62 92 82 75 46

7 11 40 39 24 74 63 62 89 92 82 75 46



Preorder: 15, 45, 26, 40, 49, 31, 11, 62, 54, 9, 28

Inorder: 26, 40, 45, 49, 15, 62, 11, 54, 31, 9, 28

Postorder: 40, 26, 49, 45, 62, 54, 11, 28, 9, 31, 15.

- Binary Search Tree
- AVL Tree

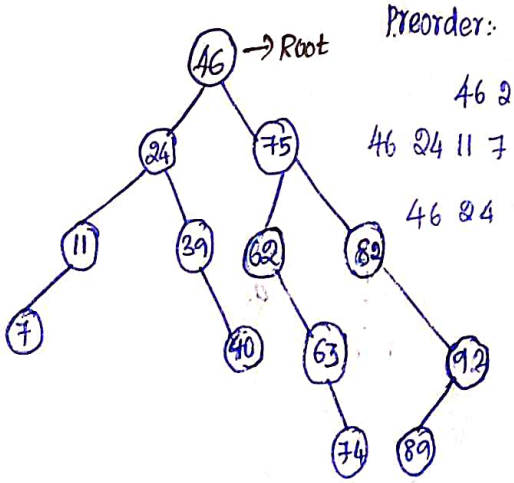
→ special Types

LST elements ≤ parent ≤ RST element.

$L \leq P \leq R$

Input sequence:

46, 75, 62, 24, 39, 11, 82, 63, 40, 7, 92, 74, 89



Preorder: 46 24 75

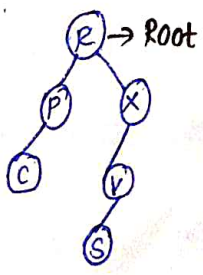
46 24 11 39 75 62 82

46 24 11 7 39 40 75 62 63 82 92

46 24 11 7 39 40 75 62

63 74 82 92 89

Input sequence: R, X, P, V, C, S



Postorder:

P, X, R

C, P, V, X, R

C, P, S, V, X, R

Preorder: R P X

R, P, C, X, V

R, P, C, X, V, S

Inorder: P, R, X

C, P, R, V, X

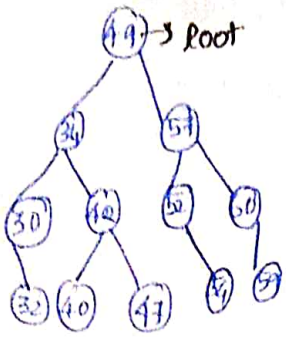
C, P, R, S, V, X

Binary Search Tree implementation

Algorithm INSERT_BST (Root, x)

```
1.  T: GETNODE()
2.  T→data = x
3.  T→lchild = NULL
4.  T→rchild = NULL
5.  If  ROOT = NULL
6.      Root = T
7.      Return
8.  Par = NULL
9.  cur = Root
10. While cur ≠ NULL
11.     if cur→data < x
12.         par = cur
13.         cur = cur→rchild
14.     else if cur→data > x
15.         par = cur
16.         cur = cur→lchild
17.     else
18.         Print "Duplicate element. Can't insert"
19.         Exit
20. End while
    End if
    If par→data < x
21.     par→rchild = T
22. else
23.     Par→lchild = T
24. Return.
```

- Finding Minimum element.
- Finding Maximum element.
- Searching for a given element.
- Deletion.



Algorithm MINIMUM (Root):-

1. If Root = NULL
2. Print "Empty BST"
3. Exit
4. cur = Root
5. While cur → lchild ≠ NULL
6. cur = cur → lchild
7. Return cur → data.

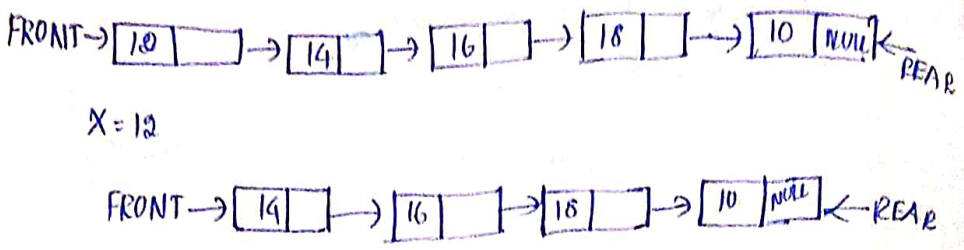
Algorithm MAXIMUM (Root):-

1. If Root = NULL
2. Print "Empty BST"
3. Exit
4. cur = Root
5. While cur → rchild ≠ NULL
6. cur = cur → rchild
7. Return cur → data.

1. Algorithm checklist (first, nodeptr).

1. If nodeptr → link = first
2. Return true
3. Else
4. Return false

2.

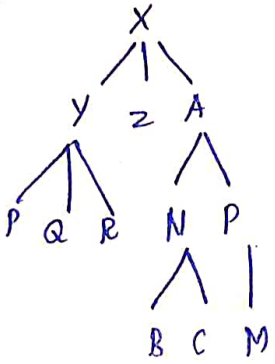


X = 12

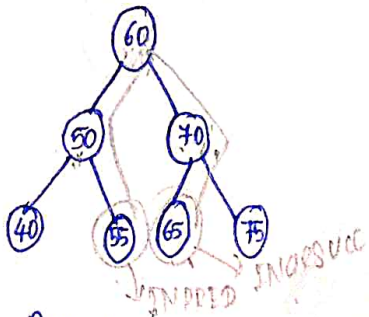
3. Degree ⇒ C → 3
D → 2

Level ⇒ B → 3
M → 4

4. (X(Y(PQR) Z A (N(BC) P(M)))

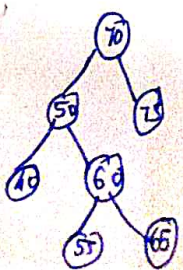


7.

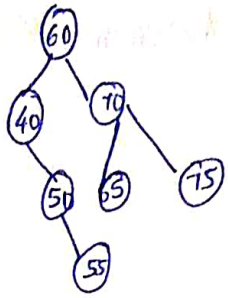


Preorder 60 50 40 55 70 65 75

70 50 60 40 55 65 75



60 40 50 55 70 65 75



8. Post Order:

40, 55, 50, 65, 75, 70, 60.

Inorder:

40, 50, 55, 60, 65, 70, 75.

9. Algorithm INPRED (x)

1. P = X → lchild
2. While P → rchild ≠ NULL
3. P = P → rchild
4. return (P).

Algorithm INSUCC (x)

1. P = X → rchild
2. While P → lchild ≠ NULL
3. P = P → lchild
4. return (P).

Algorithm BST_INSERTION_RECURSION (T,x)

1. If T = NULL
2. T = GETNODE()
3. T→data = x
4. T→lchild = T→rchild = NULL
5. return T.
6. If T→data > x
7. T→lchild = BST_INSERT_REC (T→lchild, x)
8. Else if T→data < x
9. T→rchild = BST_INSERT_REC (T→rchild, x)
10. Else
11. print " Duplicate value. can't insert"
12. Return T.

Call: BST_INSERT_RECURSION (Root, x)

40, 25, 47, 16

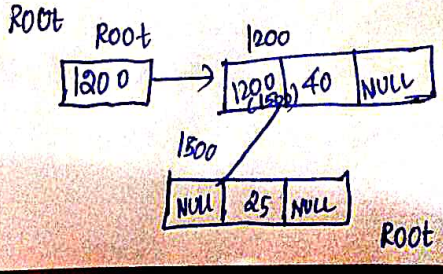
Root = NULL

Root = BIR (NULL, 40) 1200

1. T = NULL True
2, 3, 4.
5. return T.

Root = BIR (1200, 25)

1. T = NULL. False
2. 40 > 25 True
7. 1200→lchild = BIR (NULL, 25)
1. T = NULL True
2, 3, 4
5. Return T (1500)
12. Return T (1200)



Root = BIR (1200, 47)

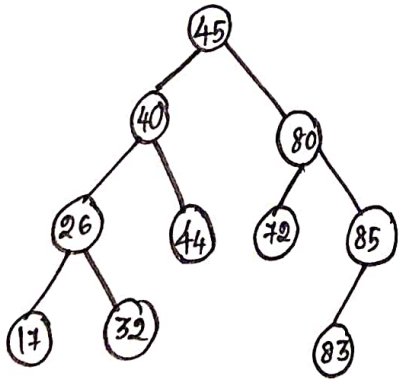
Root = BIR (Root, 47)

1. T = NULL false
6. 40 > 47 false
8. 40 < 47. True
9. Root → rchild = BST_INSERT_REC (NULL, x).
    1. T = NULL (false)
    2, 3, 4
    5. Return T. (1600)
    10. Return T.

ROOT = BIR (1200, 16)

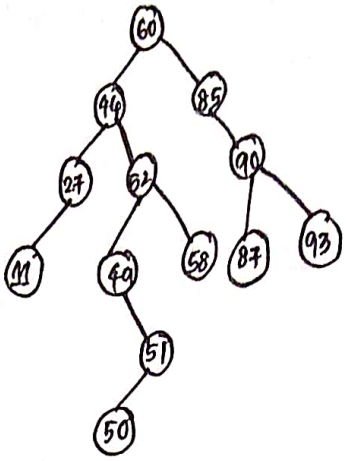
1. T = NULL false
6. 40 > 16 True
7. 1200 → lchild = BIR (1500, 16)
    1. T = NULL
    6. 85 > 16 True.
    7. 1500 → lchild = BIR (NULL, 16)
        1. T = NULL
        2, 3, 4
        5. Return T.

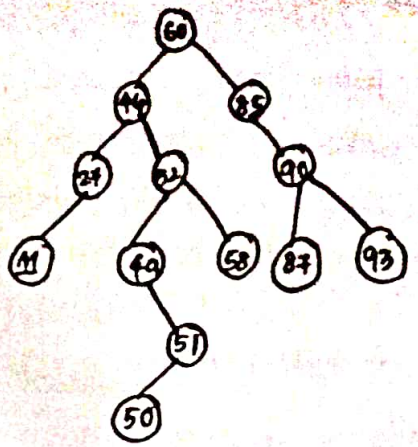
BST_DELETION :-



3 cases :-

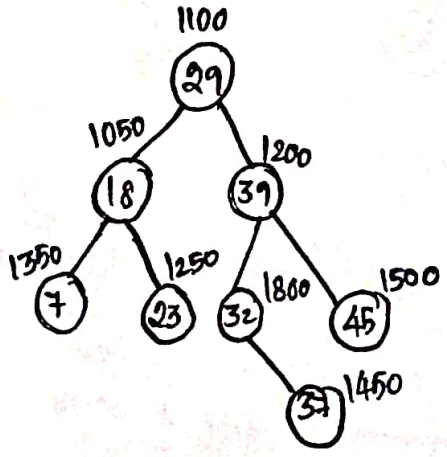
- If x is in a leaf node
- If x has single child (either left child alone or right child alone)
- If x has both children
    ↳ replaced with inorder predecessor or inorder successor.


60
94
85
27
52
90
11
40
58
87
93
51
50



Algorithm - DELETE_REC (T, x)

1. If T = NULL
2. print "Element not present in BST"
3. Return T
4. If T→data > x
5. T→lchild = BST_DELETE_REC (T→lchild, x)
6. Else if T→data < x
7. T→rchild = BST_DELETE_REC (T→rchild, x)
8. Else
9. If T→lchild = NULL
10. Return T→rchild
11. else if T→rchild = NULL
12. Return T→lchild
13. Else
14. y = BST_MINIMUM (T→rchild)
15. T→data = y
16. T→rchild = BST_DELETE_REC (T→rchild, y)
17. Return T.



For 32 → DELETION:

BDR (1100, 32)

1. 1100 ≠ NULL
4. 1100 → data 29 > 32 False
6. 29 < 32 True
7. 1100 → rchild = BDR (1200, 32)

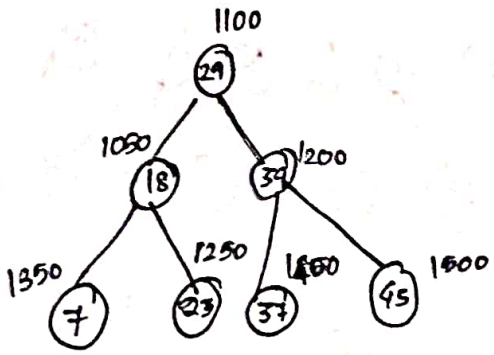
    1. 1200 ≠ NULL
    4. 34 > 32 True
    5. 1200 → lchild = BDR (1200, 32)

        BDR (1800, 32)

        1. 1800 ≠ NULL
        4. 32 > 30 False
        6. 32 < 32 False
        8. else
        9. 1800 → lchild = NULL True
        10. Return 1450.

    17. Return 1200

17. Return 1100.



Root = BDR (1100, 29)

1. 1100 ≠ NULL
2. 1100 → data(29) > 29 False
6. 1100 → data (29) < 29 False
8. Else
9. 1100 → lchild (1050) ≠ NULL
11. 1100 → rchild (1200) ≠ NULL
13. Else
14. y = BST_MINIMUM (1200) (37

15. 1100→data = 37

16. 1100→rchild = BDR (1200, 37)

1. 1200 ≠ NULL

4. 1200→data 39 > 37 True

5. 1200→lchild = NULL BDR (1450, 37).

1. 1450 ≠ NULL

4. 1450→data (37) > 37 False

6. 1450→data (37) < 37 False

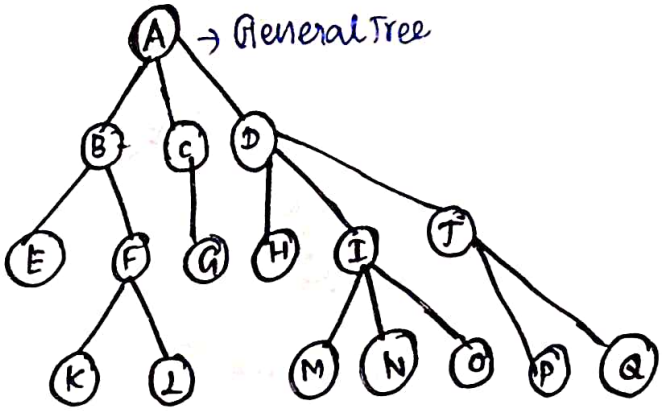
8. Else

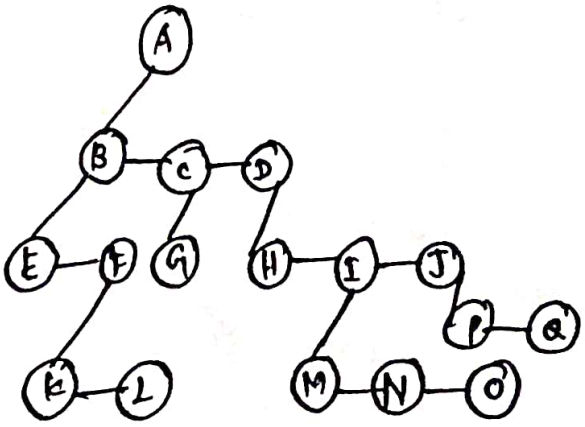
9. 1450→lchild = NULL

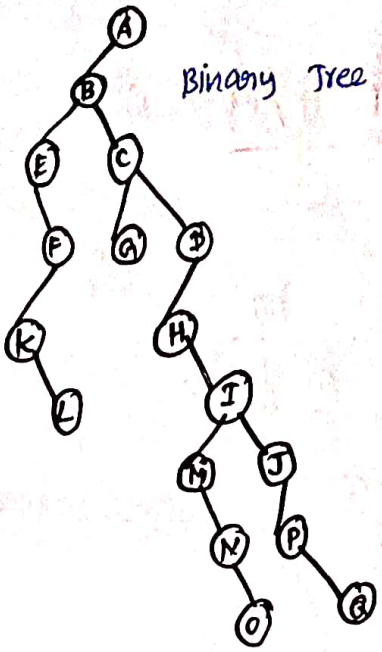
10. Return NULL.

General Tree to Binary Tree comparison:-

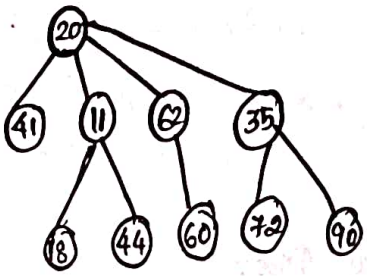
- Left child - Right Sibling Order.



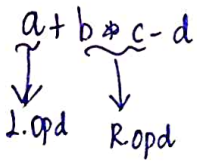


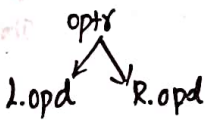


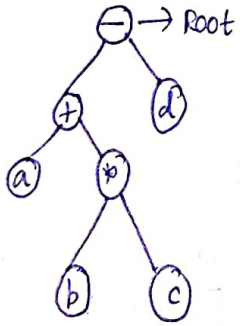
Binary Tree



Expression Trees:-







Preorder :- - + a * b c d

Postorder :- a b c * + d -

Paranthetical Representation :- ((a+(b*c)-d)

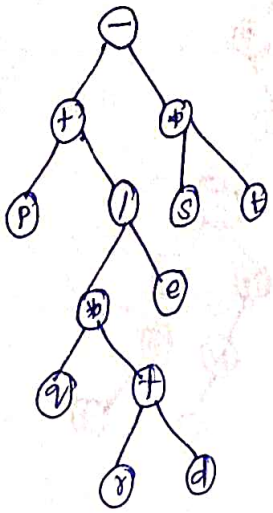
Prefix expression :- - + a * b c d

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

Postfix expression :- a b c * + d -

P+q*(r+d)/e-s*t

((((p+q*(r+d)/e)-(s*t))

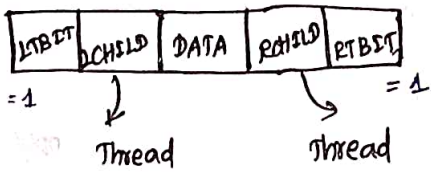


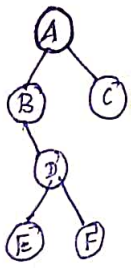
Preorder :- -+p/*q+rde*st

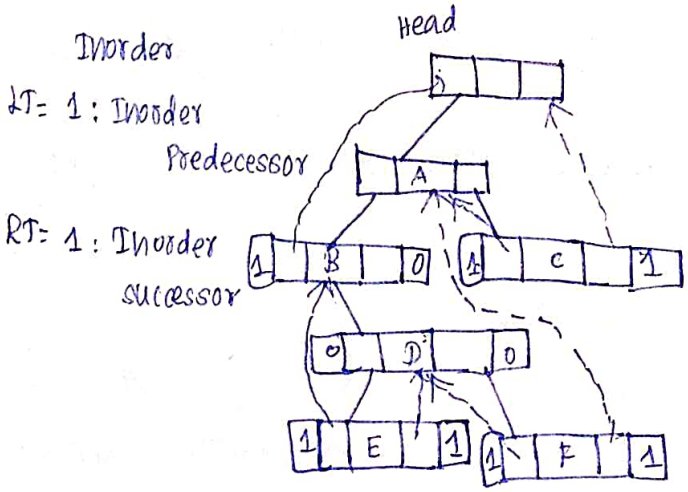
Postorder :- pqrd+*e/+st*-

Inorder :- p+q*r+d/e-s*t

Threaded Binary Tree :-







Algorithm INSUCC (T):.

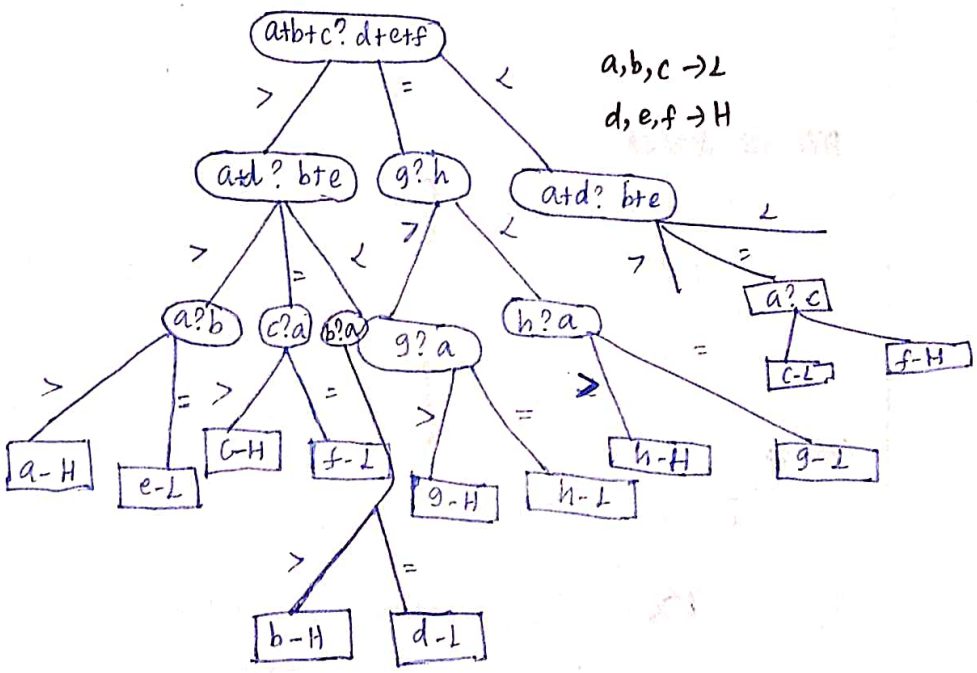
1. S = T→ rchild
2. If T→ RTBIT = 1
3. Return S
4. While S→ LTBIT ≠ 1
5. S = S→lchild
6. Return S

Algorithm INORDER (ROOT):.

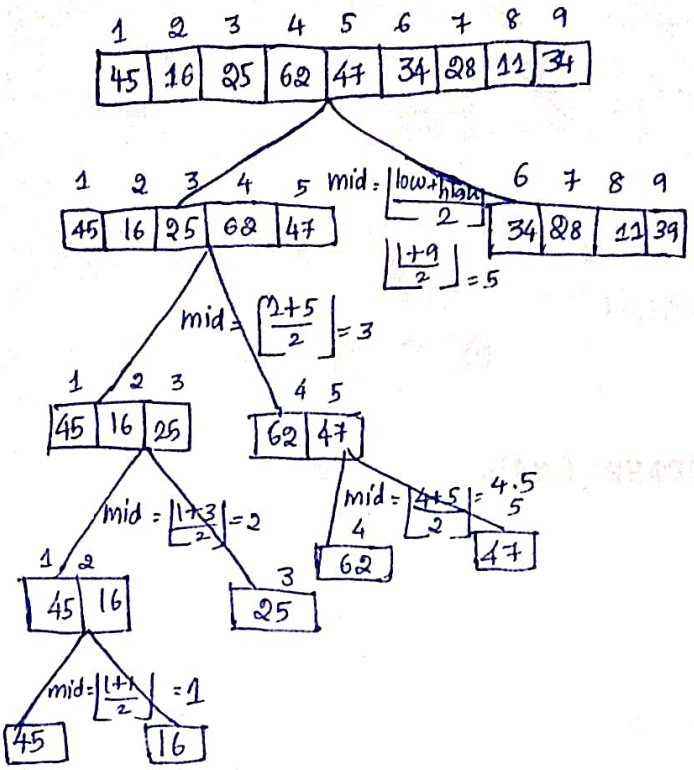
1. T = HEAD
2. Repeat
3. T = INSUCC (T)
4. If T ≠ HEAD
5. Print T→ data
6. Until T = HEAD

Decision Trees:.

8-coins a, b, c, d, e, f, g, h.

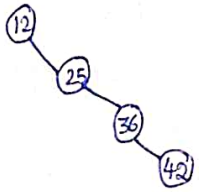


Merge sort:-



AVL Height Balanced :-

18, 25, 36, 42



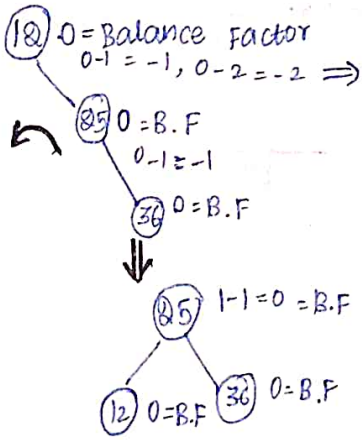
AVL is based on,

→ Height of node → [length of the longest path from current node to leaf]

→ Balance factor → [Height of left subtree − height of right subtree]

0, 1, −1

Case 1:-



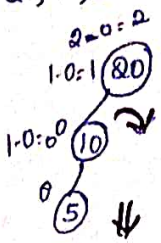
critical region

of Right child of T

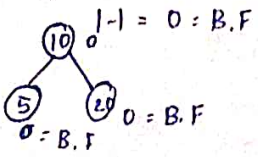
- Left Rotation

case 2: ⇒ Insertion in left subtree of left child of T.

20, 10, 5

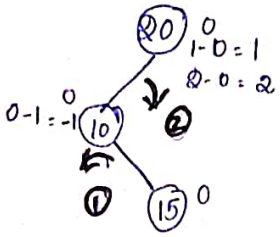


– Right Rotation.



case 3:

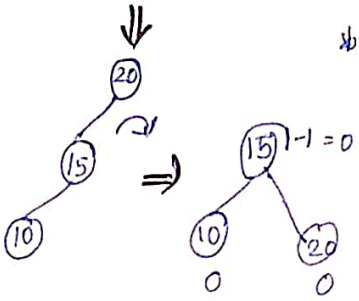
20, 10, 15



⇒ Insertion in Right subtree of left child of T

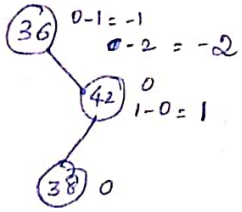
– Double rotation.

* left Rotation of leftchild of T.
* Right Rotation on T



case 4:

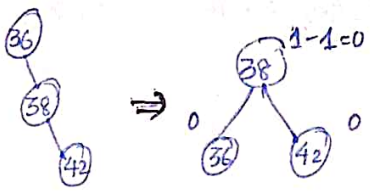
36, 42, 38



⇒ Insertion in left subtree of Right child of T.

– Double rotation

* Right rotation of Right child of T.
* Left Rotation on T.

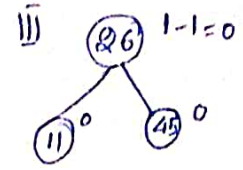


Input Sequence: 26, 11, 45, 32, 14, 19, 25, 47, 63, 54, 29, 49

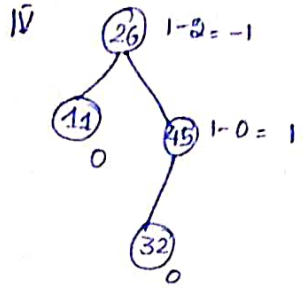
I

II

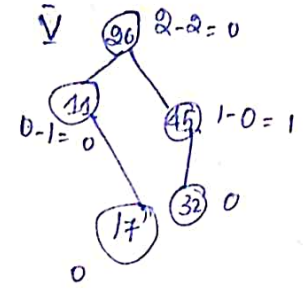
III



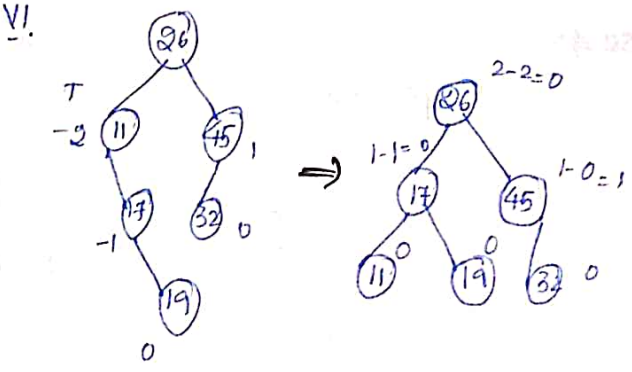
IV



V



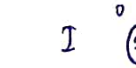
VI.



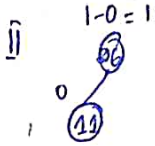
Construct AVL Tree for the following i/p sequence:

26, 11, 25, 32, 46, 31, 18, 24, 19, 54, 39, 62, 50.

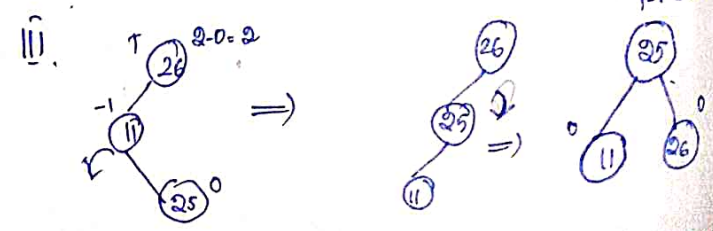
I



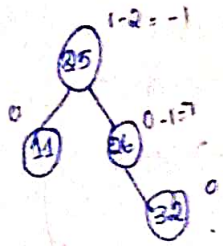
II



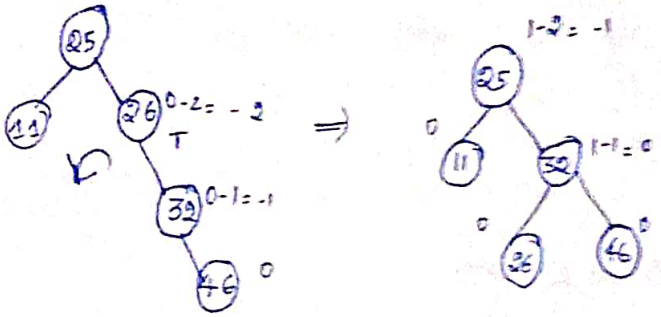
III.



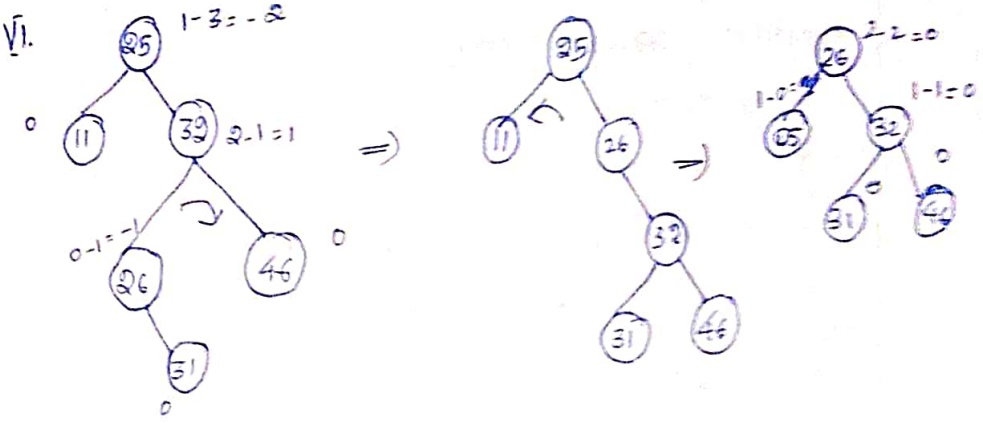
IV.



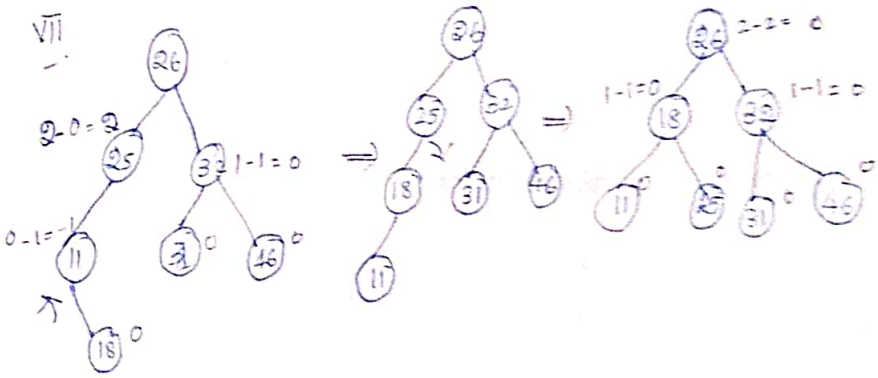
V.



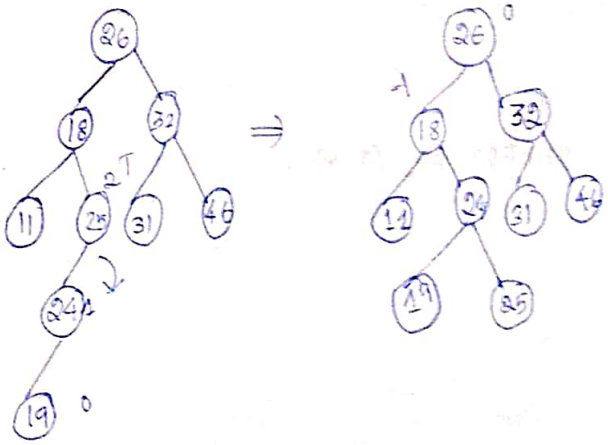
VI.



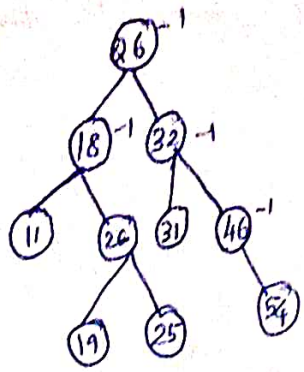
VII.



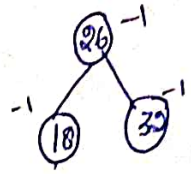
VIII.



X)



XI)



Algorithm getHeight (T):

1. If T = NULL
2. return 0
3. hl = getHeight (T→lchild)
4. hr = getHeight (T→rchild)
5. If hl ≥ hr
6. return hl+1
7. else
8. return hr+1

Algorithm getBalance Factor (T) :

1. If T = NULL
2. return 0
3. hl = getHeight (T→lchild)
4. hr = getHeight (T→rchild)
5. return hl-hr

Algorithm INSERT_AVL (T, x) :

1. If T = NULL
2. n = GETNODE ()
3. n→data = x
4. n→lchild = n→rchild = NULL
5. return n
6. If T→data > x

7. T→lchild = INSERT AVL (T→lchild, x)

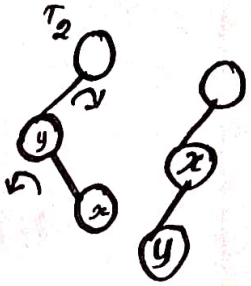
8. else if T→data < x

9. T→rchild = INSERT-AVL (T→rchild, x)

10. else

11. print "Duplicate value. Can't insert"

12. Exit.



13. bf = getBalanceFactor (T)

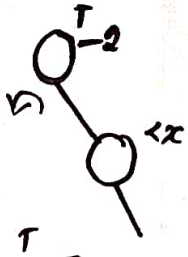
14. If bf = 2 and T→lchild →data > x

15. RIGHTROTATE (T)

16. else if bf = 2 and T→lchild → data < x.

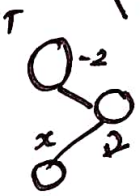
17. LEFTROTATE (T→ lchild)

18. RIGHTROTATE (T)



19. else if b = = -2 and T→lchild → data < x

20. return LEFTROTATE (T)



21. else if bf = = -2 and T→rchild→data > x

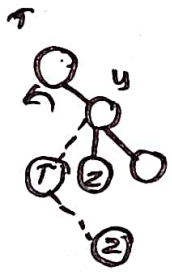
22. T→rchild = RIGHTROTATE (T→rchild)

23. return LEFTROTATE (T)

24. Return T.

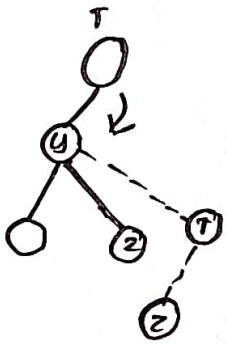
Algorithm LEFTROTATE (T):

1. y = T→rchild
2. T→rchild = y→lchild
3. y→lchild = T
4. Return y.



Algorithm RIGHTROTATE (T):

1. y = T→lchild
2. T→lchild = y→rchild
3. y→rchild = T
4. Return y.

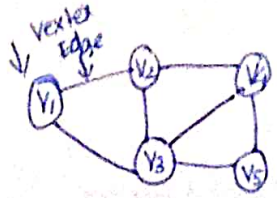


Graph::

$G = (V, E)$

$V = \{v_1, v_2 \ldots v_n\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_4) \ldots\}$

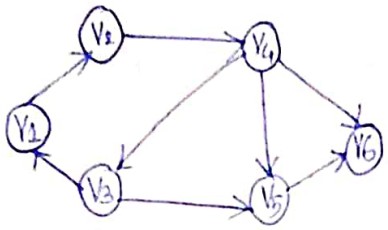


Undirected Edge
directed Edge

Undirected Graph
directed Graph.
Mixed Graph

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

$E\{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$



$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

$E = \{\langle v_1, v_2\rangle, \langle v_3, v_1\rangle, \langle v_2, v_3\rangle, \langle v_2, v_4\rangle, \langle v_4, v_3\rangle, \langle v_3, v_5\rangle,$
$\langle v_4, v_5\rangle, \langle v_4, v_6\rangle, \langle v_5, v_6\rangle\}$

(u, v)

u is adjacent to v ($\uparrow \Rightarrow$ lchild)
v is adjacent to u ($\uparrow \Rightarrow$ rchild)

$\langle u, v\rangle$

v is adjacent to u.

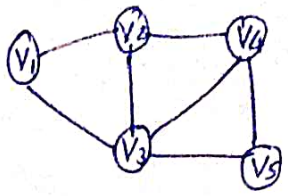
$\langle v_1, v_2\rangle$

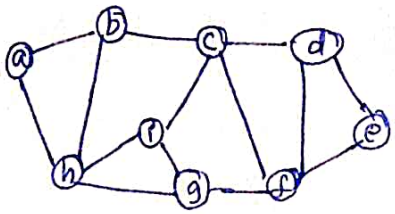
v_2 is adjacent to v_1.

Adjacent Vertices

v_1	v_2
v_2	v_3, v_4
v_3	v_1, v_5
v_4	v_3, v_5, v_6
v_5	v_6
v_6	-

Degree		Adjacent Vertices
2	V_1	V_2, V_3
3	V_2	V_1, V_3, V_4
4	V_3	V_1, V_2, V_4, V_5
3	V_4	V_2, V_3, V_5
2	V_5	V_3, V_4.





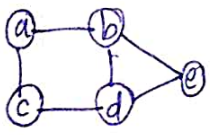
Degree		Adjacent Vertices
2	a	b, h
3	b	a, h, c
4	c	b, i, d, f
3	d	c, f, e
2	e	d, f
4	f	g, e, d, c
4	g	h, i, f
3	h	a, b, i, g
	i	c, h, g

$V = \{a, b, c, d, e, f, g, h, i\}$

$E = \{(a,b), (a,h), (b,h), (b,c),$

1. Adjacency Matrix
2. Adjacency List
3. Edge List

① Adjacency Matrix Representation :



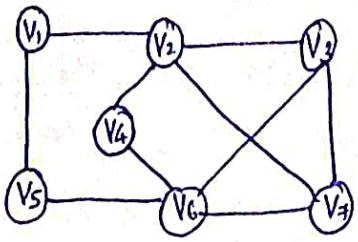
n x n

n = |V|

5 x 5

		a	b	c	d	e	Degree
		1	2	3	4	5	
a	1	0	1	1	0	0	2
b	2	1	0	0	1	1	3
c	3	1	0	0	1	0	2
d	4	0	1	1	0	1	3
e	5	0	1	0	1	0	2
		2	3	2	3	2	

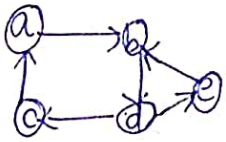
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$



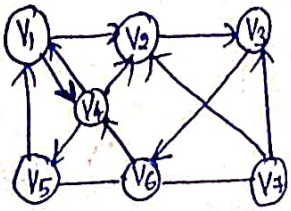
7X7

	V1	V2	V3	V4	V5	V6	V7	Degree
	1	2	3	4	5	6	7	
V1 1	0	1	0	0	1	0	0	2
V2 2	1	0	1	1	0	0	1	4
V3 3	0	1	0	0	0	1	1	3
V4 4	0	1	0	0	0	1	0	2
V5 5	1	0	0	0	0	1	0	2
V6 6	0	0	1	1	1	0	1	4
V7 7	0	1	1	0	0	1	0	3
	2	4	3	2	2	4	3	

Directed Graph:

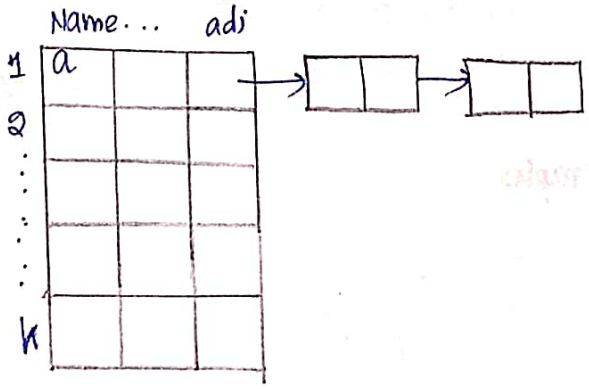


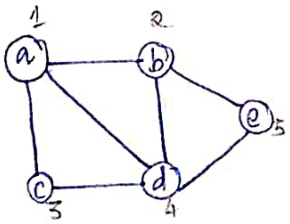
	a	b	c	d	e	Degree
	1	2	3	4	5	
1 a	0	1	0	0	0	1
2 b	0	0	0	1	0	1
3 c	1	0	0	0	0	1
4 d	0	0	1	0	1	2
5 e	0	1	0	0	0	1
	1	2	1	1	1	

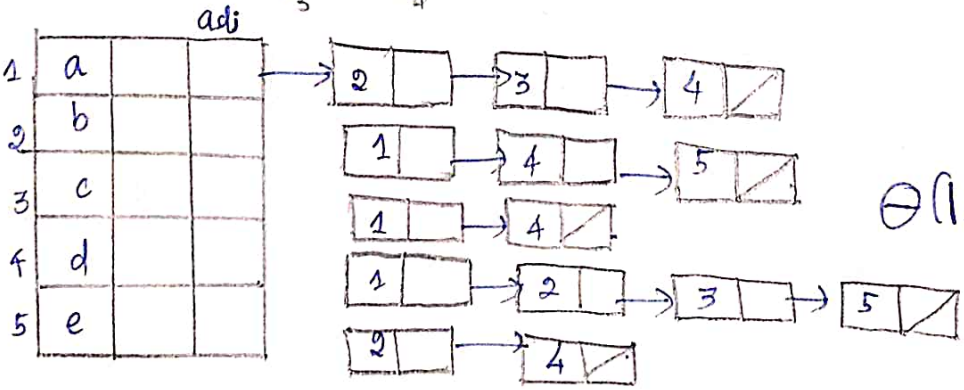


	V_1	V_2	V_3	V_4	V_5	V_6	V_7	Out Degree
V_1	0	1	0	1	0	0	0	2
V_2	0	0	1	0	0	0	0	1
V_3	0	0	0	0	0	1	0	1
V_4	1	1	0	0	1	0	0	3
V_5	1	0	0	0	0	1	0	2
V_6	0	0	0	1	0	0	0	1
V_7	0	1	1	0	0	1	0	3
In Degree	2	3	2	2	1	3	0	

②. Adjacent List Representation:

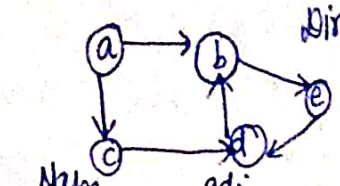




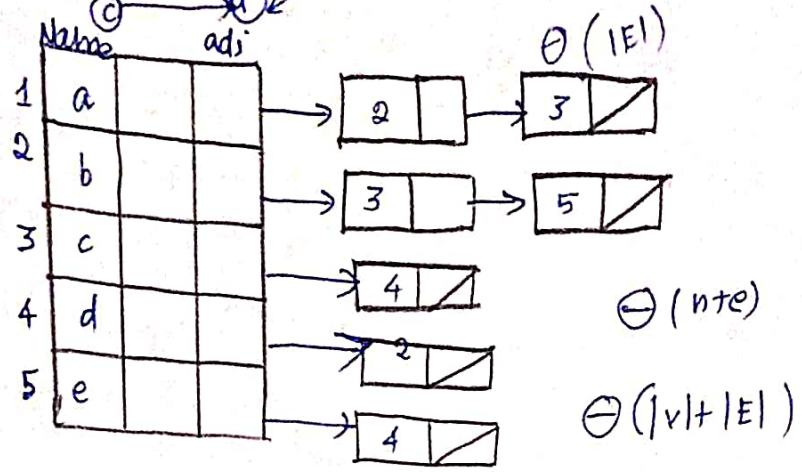


$\Theta(|E|)$

Directed graph.



	Name	adj
1	a	
2	b	
3	c	
4	d	
5	e	



$\Theta(|E|)$

$\Theta(n+e)$

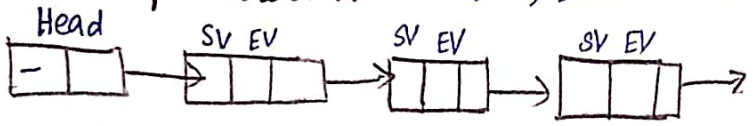
$\Theta(|V|+|E|)$

$\Theta(|V|)$

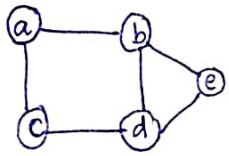
③ Edge List Representation:-

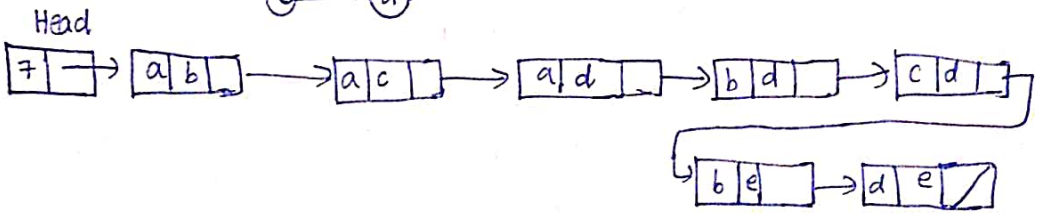
SV → starting Vertex

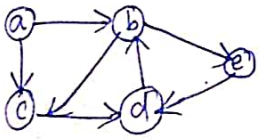
EV → Ending Vertex

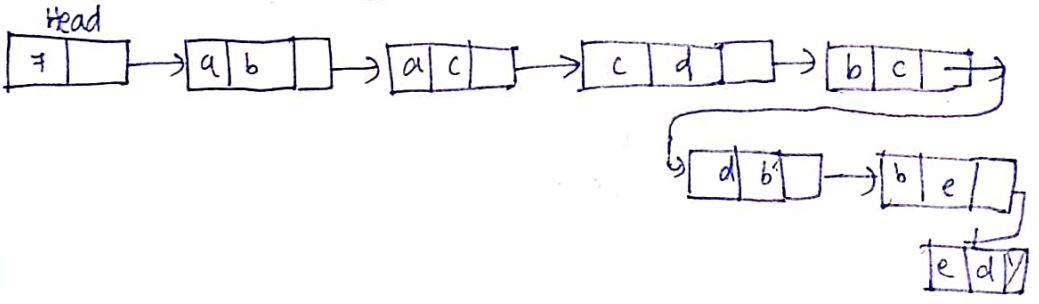


$\Theta(|E|)$

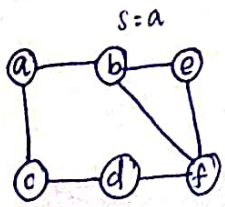








Breadth First Traversal:-

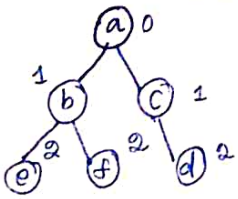


	Name	dist	visited	eptr
1	a	0	T	→ 2 → 3
2	b	1	T	→ 1 → 5 → 6
3	c	1	T	→ 1 → 4
4	d	2	T	→ 3 → 6
5	e	2	T	→ 2 → 6
6	f	2	T	→ 2 → 4 → 5

Implementation:-

```
Struct node
{
    int av_ind;
    struct node * link;
};
struct Vertex
{
    char name[10];
    int dist;
    int visited;
    node * eptr;
};
```

Breadth First Tree



struct Vertex v[10];

u = 1 2 3 5 6 4

Algorithm BFS (G, s)

1. Createqueue (Q)
2. For i = 1 to n
3. v[i].dist = ∞
4. v[i].visited = 0
5. v[s].dist = 0
6. v[s].visited = 1
7. ENQUEUE (Q, Front, Rear, s)
8. While not ISEmpty (Q)
9. u = DEQUEUE (Q, Front, Rear)
10. t = v[u].eptr
11. While t ≠ NULL
12. j = t → data

13. if V[j].visited = false

14. V[j].dist = V[u].dist + 1

15. V[j].visited = 1

16. ENQUEUE (Q, Front, Rear, j)

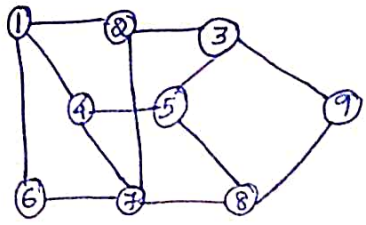
end if.

17. t = t→link

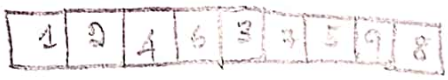
end while

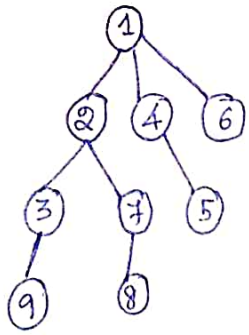
end while

18. Return.

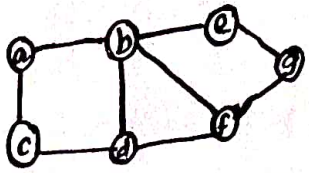


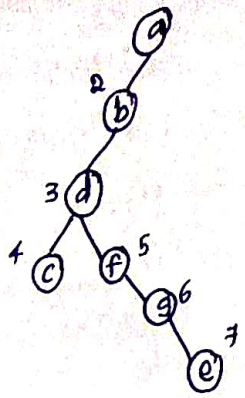
	dist	visited	ptr
1	∞/0	F/T	→ 2 → 4 → 6
2	∞/1	F/T	→ 1 → 3 → 7
3	∞/2	F/T	→ 2 → 5 → 9
4	∞/1	F/T	→ 1 → 5 → 7
5	∞/2	F/T	→ 3 → 4 → 8
6	∞/1	F/T	→ 1 → 7
7	∞/2	F/T	→ 2 → 4 → 8
8	∞/3	F/T	→ 5 → 7 → 9
9	∞/3	F/T	→ 3 → 8



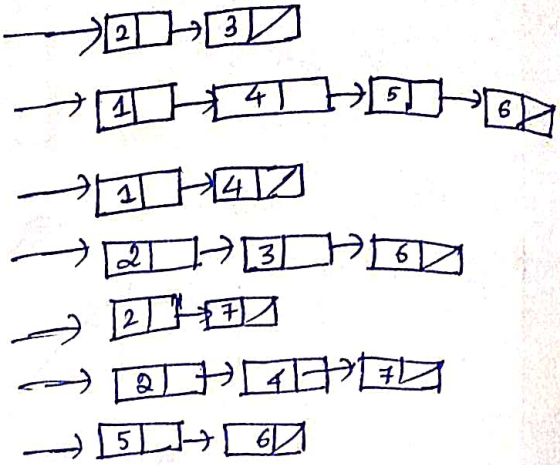


Depth First Search :-





	Name	dfn	visited	eptr
1	a	0/1	F/T	
2	b	0/2	F/T	
3	c	0/4	F/T	
4	d	0/3	F/T	
5	e	0/7	F/T	
6	f	0/5	F/T	
7	g	0/6	F/T	

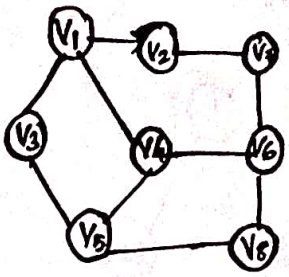


Algorithm DFS (G) :.

1. For i=1 to n
2. V[i]. visited = False
3. V[i]. dfn = 0
4. count = 0
5. For i = 1 to n
6. if V[i]. visited = False
7. DFS_VISIT(G, i)

Algorithm DFS_ VISIT (G, u):.

1. count = count+1
2. V[u]. dfn = count
3. V[u]. visited = True
4. t = eptr
5. While t≠NULL
6. w = t→ data
7. if V[w]. visited = False
8. DFS. VISIT (G, w)
9. t = t→ link
10. return

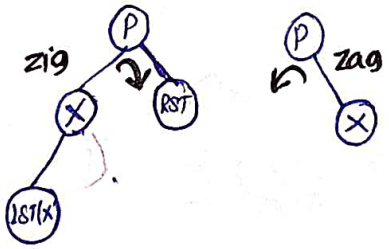


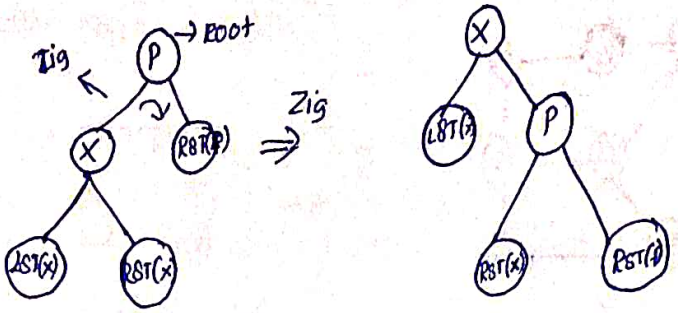
	Name	dfn	Visited	eptr
1 V_1	V_1	1	F T	$V_2 \to V_3 \to V_4$
2 V_2	V_2	2	F T	$V_1 \to V_7$
3 V_3	V_3	7	F T	$V_1 \to V_5$
4 V_4	V_4	5	F T	$V_1 \to V_5 \to V_6$
5 V_5	V_5	6	F T	$V_3 \to V_4 \to V_8$
6 V_6	V_6	4	F T	$V_4 \to V_7 \to V_8$
7 V_7	V_7	3	F T	$V_2 \to V_6$
8 V_8	V_8	8	F T	$V_5 \to V_6$

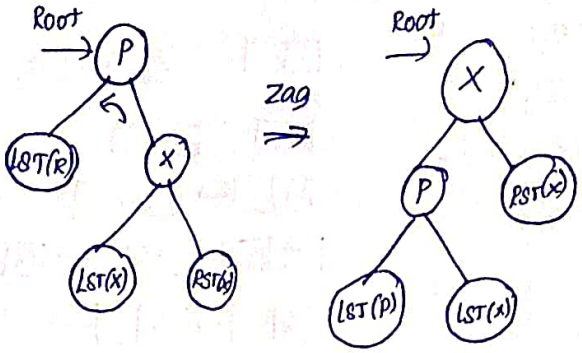
Splay Tree:-

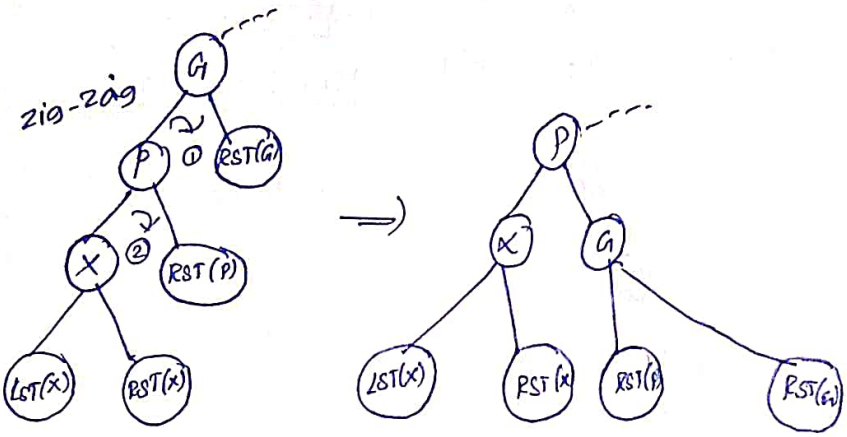
- Self adjusting BST

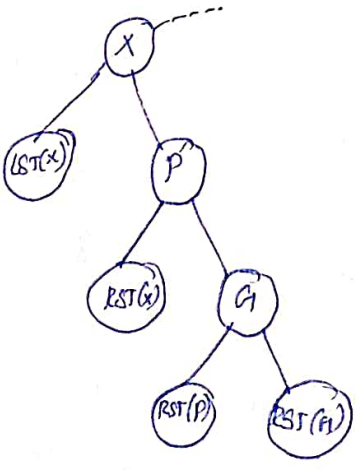
- Zig, Zag, Zig-Zig, Zag-Zag, Zig-Zag, Zag-Zig.

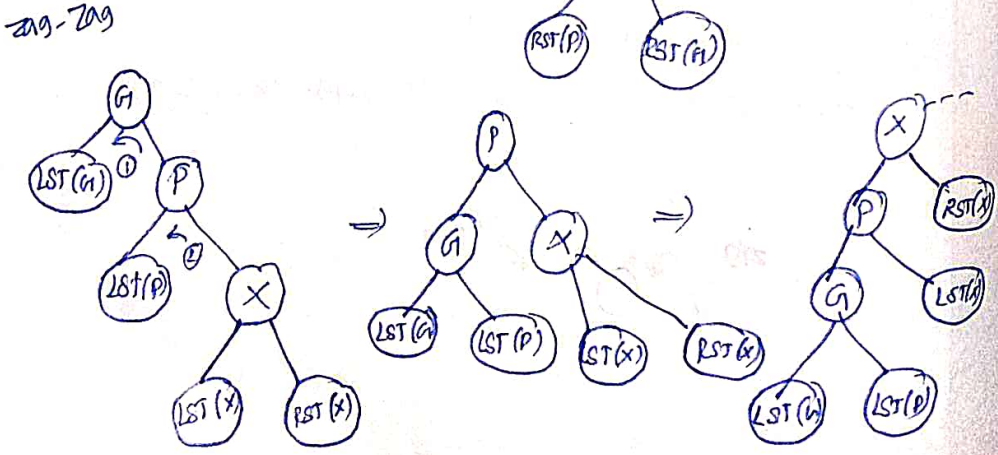



Zig
Root
P
X
RSTRP
LST(x)
RST(x)
Zig
X
LST(P)
P
RST(x)
RST(P)

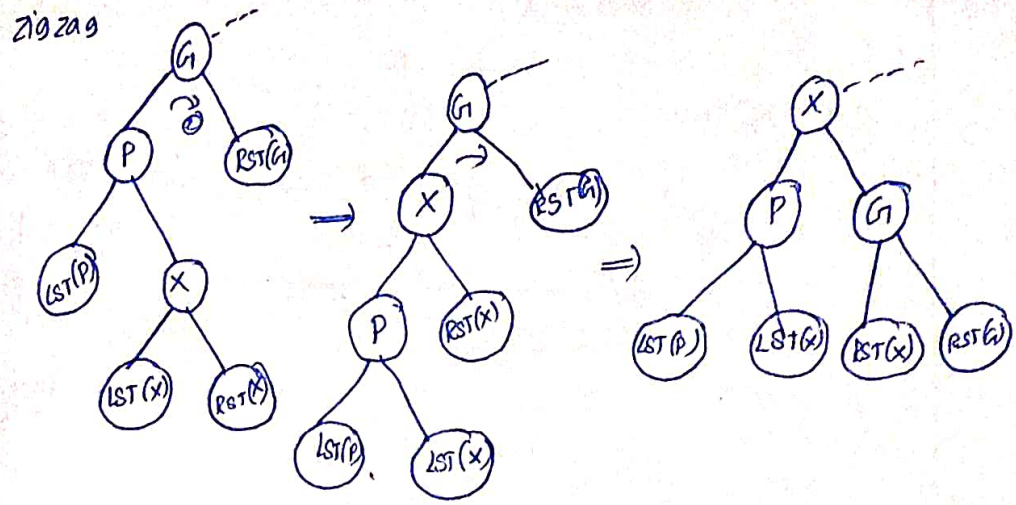

Root
P
LST(k)
X
LST(x)
RST(x)
Zag
Root
X
P
RST(x)
LST(P)
LST(x)


Zig-Zag
G
P
①
RST(G)
X
②
RST(P)
LST(x)
RST(x)
P
X
G
LST(x)
RST(x)
RST(P)
RST(G)

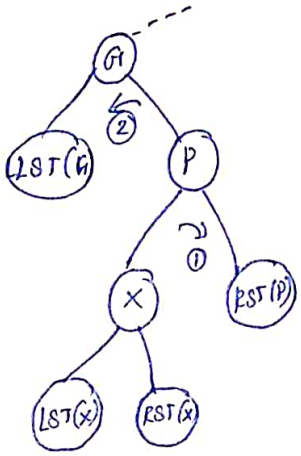

X
LST(x)
P
RST(x)
RST(P)
RST(G)


Zag-Zag
G
LST(G)
①
P
②
LST(P)
X
LST(x)
RST(x)
P
G
X
LST(G)
LST(P)
LST(x)
RST(x)
X
P
RST(x)
G
LST(P)
LST(G)
LST(P)

Zig zag



Zag-Zig



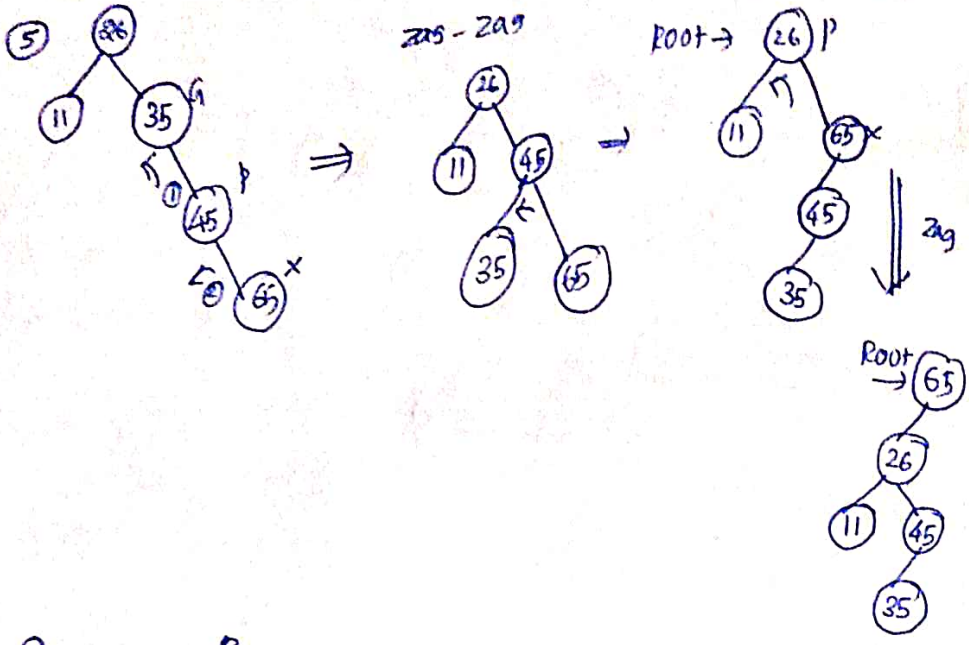
45, 11, 35, 26, 65, 90.

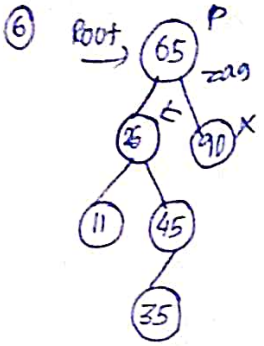
① Root → 45

② Root → 45 (P), 11 (X) ⇒ Root → 11, 45

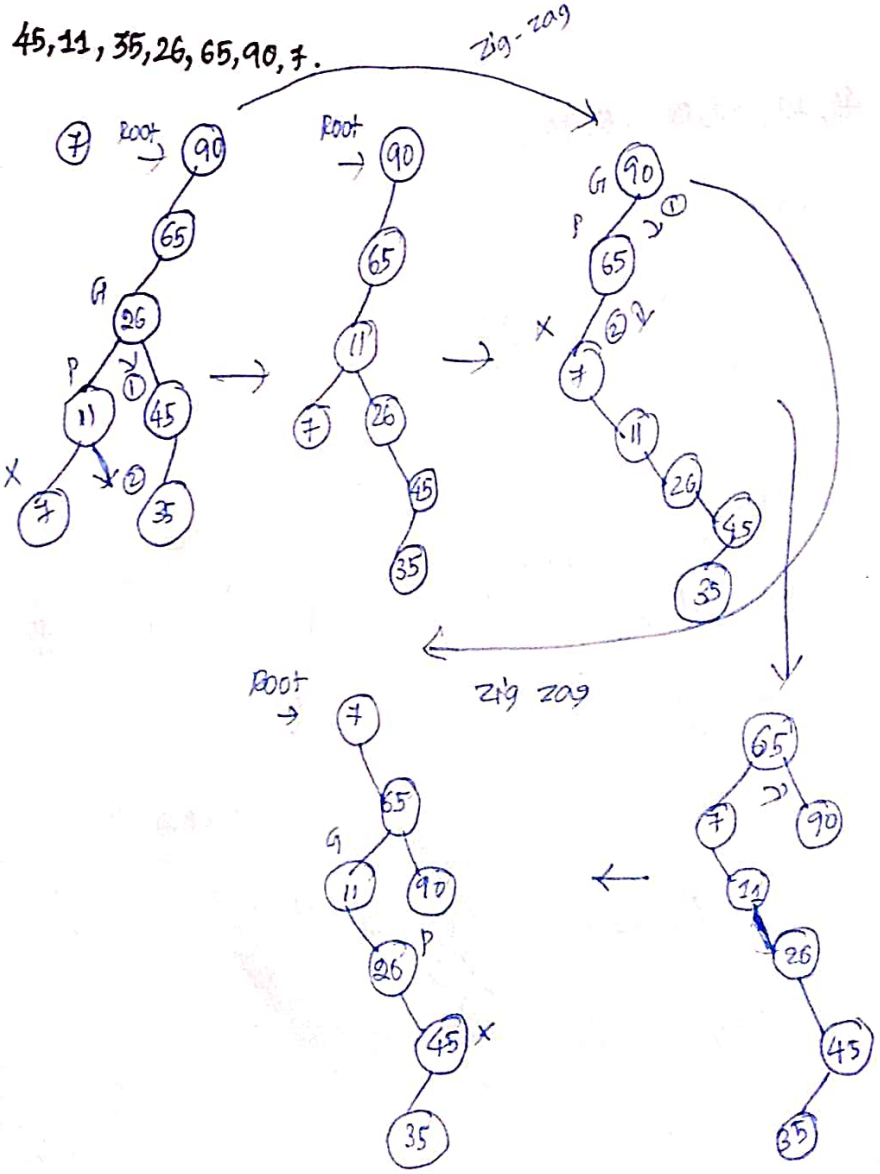
③ Root → G(11), P(45), X(35) → Zig-Zag → 14, 35, 45 → Root 35, 11, 45

④ Root → 35 (G), P(11), 45, X(26) → Zig Zag ⇒ 35, 26, 45, 11 → Root → 26, 11, 35, 45





45, 11, 35, 26, 65, 90, 7.



B-Tree :

- Strictly Balanced Tree

  M-ary Tree

  m-1 keys → Increasing Order