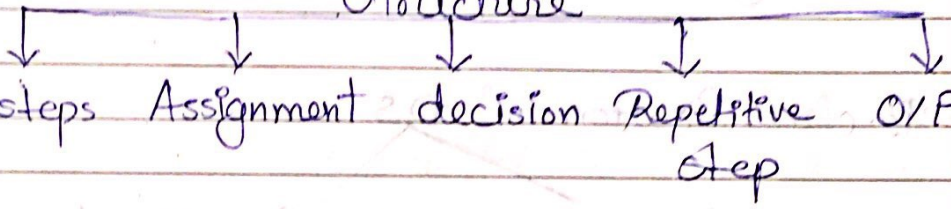
- Base address:- Address of first element of an array is known as Base address

- Algorithm: An algorithm may be defined as a finite sequences of instructions each of which has a clear meaning & can be performed in finite amount of time.

* Structure



* Properties of Algorithm:

(1) Finiteness: An Algorithm must terminate after a finite number of steps.

(2) Definiteness: The steps of the Algorithm must be precisely define or unambiguous specified. (clearly defined)

(3) Generality: An Algorithm must be generic enough to solve all problems of particular class.

(4) Effectiveness: The operations of the algorithm must be basic enough to be put down on pencil & paper.

(5) Input / output: The algorithm must have certain initial & precise inputs & outputs that may be ger

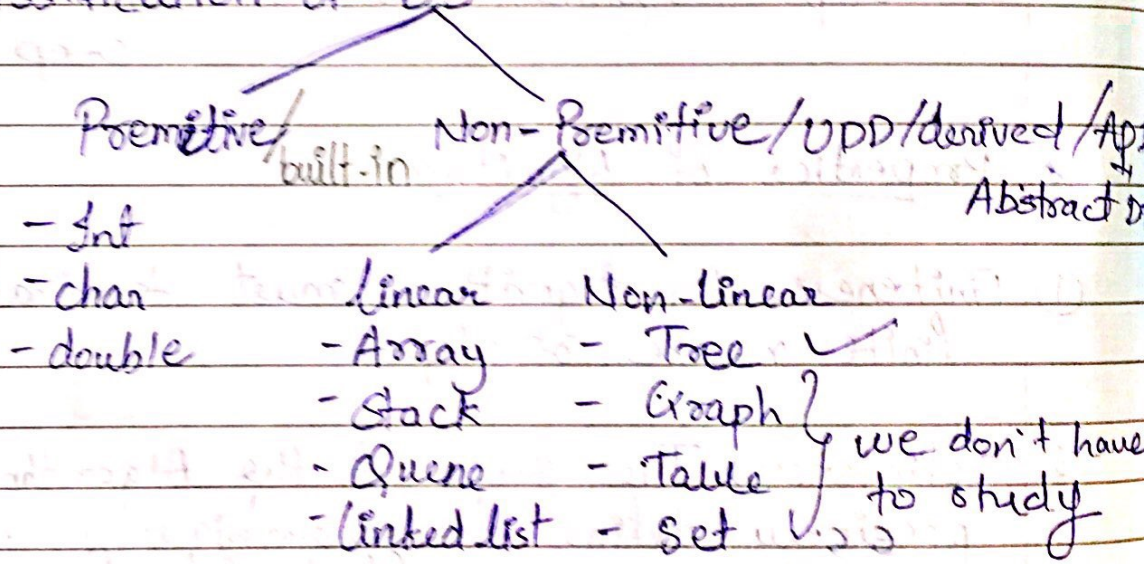
both intermediate & final steps -

Data structure + Algorithm:

- Data structure: A specialized format for organizing & storing data is known as data structure.

Efficiency of Algorithm:

Q. classification of DS



Printf(a[i]);
Printf(i[a]); output will be same bcuz when we declare array a[5] it will create base address & when declare i[a] that time also it will take base address so....

- Abstract Data type:
  - When an application requires a special kind of data which is not available as a built-in data type then it is programmer responsibility to implement his own kind of data. Here the programmer has to specified how to stored a value for that data, what are the operations that can be performed & what amount of memory is required to stored the data.

  - Programmer's own datatype is termed as abstract data type which is also known as user-defined data type.

- Static Memory: When memory allocation at compile time is known as---

- Dynamic Memory: When memory allocation at run-time is known as---

- Array: An array is a finite, ordered & collection of homogenus (similar type of datatype) data elements.

- Array
  - size
  - Type
  - Base Address
  - Index
  - Range of Indices (lower bound, upper bound)
  - word size

int a[5] is called as one-deminsional or subscript

a[i] index

i, j, k is called counterd variable

- lower bound: when the array gets starts it is known
- upper bound: " " " " ends " "

* ~~Address~~
* Formula:

Base address, lower bound

- Address A[i] = B + (i - L) * w ← word size

- Index (Ai) = L + i - 1

upper bound

- Size (A) = U - L + 1

* Example:

- Suppose an array ~~A[-15 -~~ A[-15...64] is stored in a memory whose starting Address is 459 assume that the word size ~~for~~ each element is 4. Then obtain the following:

① How many number of elements are there in the array A?

② How much memory requires to store the entire array?

③ What is the location for A[50]?

④ What is the location of the 10th element?

⑤ Which element is located at ~~559~~ 431?

⇒ (1) (Ai) = -15+0-1 Size = U-L+1

= -15-1 = 64+15+1

= -16 = 80

⇒ (2) size(A) = 64-(-15)+1

= 64+15+1

= 79+1

= 80

⇒ (3) Address A[50] = 459 + (50-(-15)) * 4

= 459 + (65) * 4

= ~~444 * 4~~ ~~624 * 4~~ 459 + 260

= ~~1896~~ ~~2096~~ 719

⇒ (4) Index (Ai) = -15+10-1

= -5-1

= [-6] = 459 + (-6-(-15)) * 4

= 459 + (-9) * 4

= 459 - 36

⇒ (5) ~~Address A[i] = 459 + (0-15) * 4~~

= ~~479~~ 731 + (-15) * 4

= ~~604 * 4~~ = 731 + 60

= ~~2416~~ = 791

Address A[i] = 731 +

* Suppose an array A ~~with~~ [-39...39] is stored in a memory whose starting address is 1369. Assume that it is a character array obtained the following.

① How many numbers of element are there in the array A?

② How much memory is required to store the entire array?

③ What is the location of A[0]?

④ What is the location of 20th element?

⑤ Which element is located at 1379?

⇒① size = U-L+1
= 39-(-39)+1
= 39+39+1
= 39+40
= 79

⇒② memory = size * w
= 79 * 1
= 79

(3) Index = Address = 1369 + (3 − (−39)) * 1
= 1369 + (3139) * 1
= 1369 + 42
= 1411

(4) Index = −39 + 20 − 1
= −39 + 19
= −20

Address = 1369 + (20 − (−39)) * 1
= 1369 + (−20 + 39) * 1
= 1369 + 19
= 1388

(5) Address = 1369 − 1349 + (i − (−39)) * 1
⟹
1379 − 1369 = (i + 39) * 1
10 = (i + 39)
10 − 39 = i
−29 = i

i	C	S
i=0	0<5	i++
1	1<5	i++
2	2<5	i++
3	3<5	i++
4	4<5	i++
5	5<5	X

Date:

* Data.

* Operations of an Arrays:

- Insertion
- Deletion
- Traversal
- Searching
- Sorting
- Merging

* Algorithm: Traversal

(1) step-1 $i = L$

step-2 while $i \leq U$ do

Step-3 Process A[i]

Step-4 $i = i + 1$

Step-5 End while

Step-6 stop

* Sample Calculations:

i	L	U	condition	
0	0	4	$0 \leq 4$	True
1	0	4	$1 \leq 4$	True
2	0	4	$2 \leq 4$	True
3	0	4	$3 \leq 4$	True
4	0	4	$4 \leq 4$	True
5	0	4	$5 \leq 4$	False

* Searching:

- Input: the key element which is to be search.

- Output: location of key element.

- Data structure: array A[L..U]

* Algorithm: Searching

Step-1 i=L, found=0, location=-1

step-2 while i≤U and found=0 do

Step-3 if compare(a[i], key)=True then

step-4 found=1

step-5 location=i

step-6 else

step-7 i=i+1

step-8 end if

step-9 end while

step-10 if found=0

step-11 print "key not found"

step-12 else

step-13 Print "key found at", location

step-14 end if

step-15 return (location)

step-16 Stop

* Deletion:

- Input: the element which is to be deleted.
- output: ~~sl~~ slimp array with key.
- data structure: array a[L..U].

* Algorithm: Deletion.

step-1 i:=searchArray (A, key)

step-2 if (i==-1) then

step-3 print "Key not found, No deletion"

step-4 Exit

step-5 else

Step-6 while i<U do

Step-7 A[i] = A[i+1]

Step-8 i = i+1

Step-9 End while

Step-10 End if

Step-11 A[U] = NULL

Step-12 Stop

* Sample Calculation:

i	i = -1	i < U	A[i] = A[i+1]	i = i+1
~~3~~	3 = -1 X	3 < 6 ✓	A[3] = A[3+1] A[3] = A[4]	3+1
4	~~4 = -1~~	4 < 6	A[4] = A[4+1] A[4] = A[5]	4+1
5		5 < 6	A[5] = A[5+1] A[5] = A[6]	5+1
6		6 < 6 X		

A[U] = A[6] : NULL

* Sorting:

- Input: An array with integer data.
- Output: An array with sorted elements.
- Data structure: Array A[L..U].

* Algorithm:

Step-1 i = U

Step-2 while i >= L do

Step-3 j = L

Step-4 while j < i do

Step-5 if order(A[j], A[j+1]) = False then

Step-6 swap(A[j], A[j+1])

Step-7 end if

Step-8 j = j+1

Step-9 end while

Step-10 i = i-1

Step-11 end while

Step-12 Stop

- This sorting method is known as bubble sort

3	15	33	79	89

* Sample calculation:

33 3 15 79

A	33	79	3	15	89

$i=U$	$i>=L$	$j=L$	$j<i$	$A[j], A[j+1]$	$A[1], A[0]$
4	$4>=0$	0	$0<4$	$A[0], A[0+1]$	
			$1<4$	$A[1], A[1+1]$	
				2	
			$2<4$	$A[2], A[2+1]$	$A[3], A[2]$
			$3<4$	$A[3], A[3+1]$	$A[4], A[3]$
			$4<4$ ✗		

33	79	3	15	89

3	$3>=0$	0	$0<3$	$A[0], A[0+1]$	
			$1<3$	$A[1], A[1+1]$	$A[2], A[1]$
				2	
			$2<3$	$A[2], A[2+1]$	$A[3], A[2]$
				3	
			$3<3$ ✗		

33	3	15	79	89

2	$2>=0$	0	$0<2$	$A[0], A[0+1]$	$A[1], A[0]$
			$1<2$	$A[1], A[1+1]$	$A[2], A[1]$
			$2<2$ ✗		

3	15	33	79	89

1	$1>=0$	0	$0<1$	$A[0], A[0+1]$	✓
		1	$1<1$ ✗		

0 $0 \geq 0$ $0 < 0$ ✗

Merging:

Step-1 $i=0$, size $= (U_1 + U_2) + 1$, A[size], $j=0$

Step-2 while $i <$ size do

Step-3 if ($i < U_1$)

Step-4 A[i] = B[i]

Step-5 else

Step-6 A[i] = C[i]

Step-7 $j = j + 1$

Step-8 end if

Step-9 $i = i + 1$

Step-10 end while

Step-11 stop

Merging:

- Input: two arrays $A_1[l_1..u_1]$, $A_2[l_2..u_2]$
- Output: resultant array $A[l..u]$.
- Data structure: $A[l..u]$.

Algorithm:

Step-1 $i_1 = l_1$, $i_2 = l_2$

Step-2 $l = l_1$, $u = u_1 + u_2 + 1$

Step-3 $i = l$

Step-4 Allocate memory (size $(u-l+1)$)

Step-5 while $i_1 \leq u_1$ do

Step-6 $A[i] = A_1[i_1]$

Step-7 $i = i+1$, $i_1 = i_1 + 1$

Step-8 end while

Step-9 while $i_2 \leq u_2$ do

Step-10 $A[i] = A_2[i_2]$

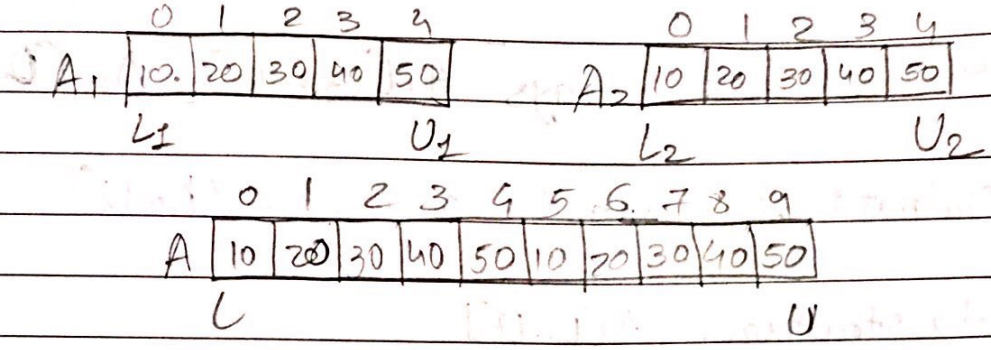
Step-11 $i + i+1$, $i_2 = i_2 + 1$

Step-12 end while

Step-13 Stop

Page No.:
Date:

Sample calculation:



$i_1 = L_1$	$i_2 = L_2$	$L = L_1$	$U = U_1 + U_2 + 1$	$i = L$	$U - L + 1$
0	0	0	4+4+1=9	0	9-0+1 = 10

$i_1 \leq U_1$	$A[i] = A_2[i_2]$	$i = i + 1$	$i_1 = i_1 + 1$
$0 \leq 4$	$A[0] = A_2[0] = 10$	0+1=1	0+1=1
$1 \leq 4$	$A[1] = A_1[1] = 20$	1+1=2	1+1=2
$2 \leq 4$	$A[2] = A_1[2] = 30$	2+1=3	2+1=3
$3 \leq 4$	$A[3] = A_1[3] = 40$	3+1=4	3+1=4
$4 \leq 4$	$A[4] = A_1[4] = 50$	4+1=5	4+1=5
$5 \leq 4$ X			

$i_2 \leq U_2$	$A[i] = A_2[i_2]$	$i = i + 1$	$i_2 = i_2 + 1$
$0 \leq 4$	$A[5] = A_2[0]$	5+1=6	0+1=1
$1 \leq 4$	$A[6] = A_2[1]$	6+1=7	1+1=2
$2 \leq 4$	$A[7] = A_2[2]$	7+1=8	2+1=3
$3 \leq 4$	$A[8] = A_2[3]$	8+1=9	3+1=4
$4 \leq 4$	$A[9] = A_2[4]$	9+1=10	4+1=5
$5 \leq 4$ X			

2D / double subscripted array:

- 2-dimensional arrays are a collection of homogenous elements where the elements are ordered in a number of rows & columns.

e.g: $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}_{m \times m}$

- Two methods to store 2-D in a memory:
  (i) ~~row-by~~ row-major order
  (ii) column-major order

row	column
1	1
2	4
3	7
4	2
5	5
6	8
7	3
8	6
9	9

- Formula for row-major order =

Address A[i][j] = B + ((i-1) * n + j - 1) * w

(B: Base address, w: word size, i: row, n: no. of columns, j: cols)

- Formula for column-major order =

Address A(i,j) = B + ((j-1) * n + i - 1) * w

$a_{22} = 1000 + ((2-1) * 3 + 2 - 1) * 4$
$= 1000 + ((1) * 3 - 1) * 4$
$= 1000 + (3 - 1) * 4$
$= 1000 + (2) * 4$
$= 1000 + 16$
$= 1016$

$a_{22} = 1000 + ((2-1) * 3 + 2 + 1) * 4$

$= 1000 + (1 * 3 + 1) * 4$

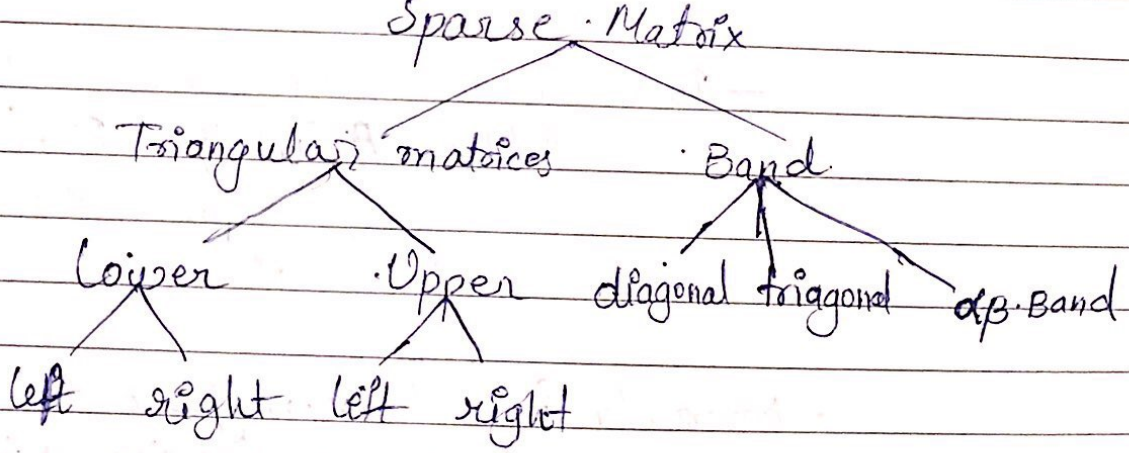
$= 1000 + (3 - 1) * 4$

$= 1000 + (4) * 4$

$= 1000 + 16$

$= 1016$

* Sparse matrix:



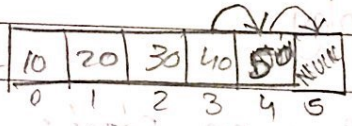
Insertion in Array:

Input: key is the item & location is index of element where it is should be inserted. output: array enrich with key.

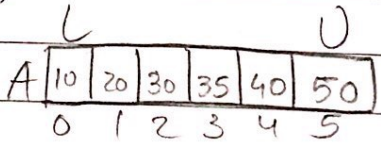
DS: A[L..U]

Step-1 If A[U] ≠ NULL then
Step-2 print "Array is full: No insertion"
Step-3 Exit
Step-4 Else
Step-5 i = U
Step-6 while i > location do
Step-7 A[i] = A[i-1]
Step-8 i = i-1
Step-9 End while
Step-10 A[location] = key
Step-11 End if
Step-12 Stop

Sample calculation:



i = U	key	location	i > location	A[i] = A[i-1]
5	35	3	5 > 3	A[5] = A[4]
			4 > 3	A[4] = A[3]
			3 > 3	X



Ch:

Stack:

- LIFO (Last In First Out).
- Stack is an ordered collection of homogenous elements where insertion & deletion operation place at one end only.
- Insertion operation is known as Push
- deletion operation is known as pop.
- Initial value of Top value = 0.
- Push
- Pop
- status
- display

Operations on stack:

- Push
- Pop
- Status
- Display

(1) Push Operation:

Input:

Output:

DS:

Algorithm:

Step-1 If top ≥ size then
Step-2 print "Stack is full"
Step-3 else
Step-4 top = top + 1
Step-5 Stack[top] = item
Step-6 End if
Step-7 Stop

5	50	top
4	40	top
3	30	
2	20	top
1	10	top

top = 0

(2) Pop:

Algorithm:

Step-1 If top = 0 then
Step-2 print "Stack is empty"
Step-3 else
Step-4 item = stack[top]
Step-5 top = top - 1
Step-6 end if
Step-7 Stop

- If my top ≥ size then my stack is stackover flow or when top is at size
- When top is at zero stack is stackunder flow.

* Status:

* Algorithm:

```
Step-1  If top=0 then
Step-2      print "Stack is empty"
Step-3  else
Step-4      if top >= size then
Step-5          print "stack is full"
Step-6      else
Step-7          freespace = size - top
Step-8          print "Available space in stack is:" freesp
Step-9      end if
Step-10 end if
Step-11 Stop
```

* Display:

* Algorithm:

```
Step-1  If top >= size then
Step-2      print "stack is full"
Step-3  else
Step-4      i=top
Step-5      while (i>0) do
Step-6          print stack[i]
Step-7          i=i-1
Step-8      end while
Step-9  end if
Step-10 stop
```

( )
^
* / . left to right
+ -

* Application of stack:

- Evaluation of Arithmetic ~~operation~~ Expression
- Implementation of Recursion.
- Factorial Calculation.
- Tower of Hanoi problem.
- String reversion

* Infix to Postfix:

* Polish Notation:

- Infix
- Postfix
- Prefix

* Infix to Prefix:

e.g: (i) a + b * c
a + * bc
+ a * bc

(ii) a * b * c + d
* ab * c + d
* * abc + d
+ * * abcd

(iii) a + b * c * d - e
a + * bc * d - e
a + * * bcd - e
+ a * * bcd - e
+ a * * bcde

(iv) A + B * C / D - E ^ F * G
A + B * C / D - ^ EF * G
A + * BC / D - ^ EF * G
A + / * BCD - ^ EF * G
A + / * BCD - * ^ EFG
+ A / * BCD - * ^ EFG
+ A / * BCD * ^ EFG

Highest priority → ( )
then → ^
then → * / L to R
then → + - L to R

$A + (B - C) * D + (E - F)$

$A + -BC * D + -EF$

$A + *-BCD + -EF$

$+A*-BCD + -EF$

$++A*-BCD-EF$

$(A+B) \wedge C - ((D*E)/F)$

$(A+B) \wedge C - (*DE/F)$

$(+AB \wedge C - /*DE$

$\wedge +ABC - /*DE$

$-\wedge +ABC/*DE$

$((A+B)*((C/D) - (E \wedge (F*G))))$

$\{+AB * (/CD\} - \{\wedge E(*FG))\}$

$\{+AB * (/CD - *\wedge EFG)\}$

$\{+AB * (-*/CD\wedge EFG)\}$

$\{*+AB - */CD\wedge EFG\}$

Infix to Postfix:

e.g: A+B+C

AB+ +C

AB+C+

A+B+C

A+B*C+D

AB+*C+D

AB+C*+D

AB+C*D+

(i) A+B*C/D-E^F*G

A+B*C/D-EF^*G

A+BC*/D-EF^*G

A+BC*D/-EF^*G

A+BC*D/-EF^G*

ABC*D/+-EF^G*

ABC*D/+EF^G*-

(ii) A+(B-C)*D/(E-F)

A+(BC-)*D/(EF-)

A+BC-D*/EF-

A+BC-D*EF-/

ABC-+D*EF-/

(iii) ((A+B)*(C/D)-(E^(F*G))))

AB+*(CD/-(E^(FG*)))

AB+*(CD/-EFG*^)

AB+*(CD/EFG*^-)

AB+CD/EFG*^-*

(iv) ((A+((B^C)-D)) * (E-(A/C)))

(A+(BC^-D)) * (E-AC/)

(A+BC^D-) * EAC/-

ABC^D-+ * EAC/-

ABC^D-+EAC/-*

* Algorithm to convert Infix to Postfix:

Step-1 Scan the symbol of infix array one-by-one from left to right

Step-2 If symbol is left '(' then add it to stack

Step-3 If symbol is operand then add it to postfix array

Step-4 (a)- If symbol is operator then pop the operators which have same precidence or high precidence than the operator which occured

(b)- Add pop operators to postfix array

(c)- Add the scan symbol operator into OPT st

Step-5 (a) If symbol is right ')' then pop all the operat from OPT stack until '(' in OPT stack.

(b)- Remove left '(' from OPT stack.

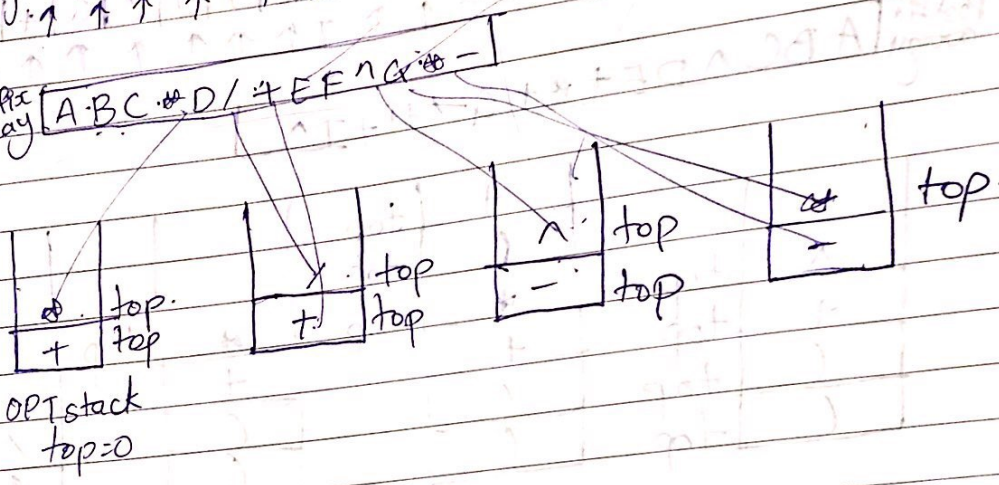
Step-6 When there is no symbol remains in infix array then pop all the symbols from OPT stack

, & add them to postfix array :Eg2

- E.g:

Infix array: A | + | B | * | C | / | D | - | E | ^ | F | * | G

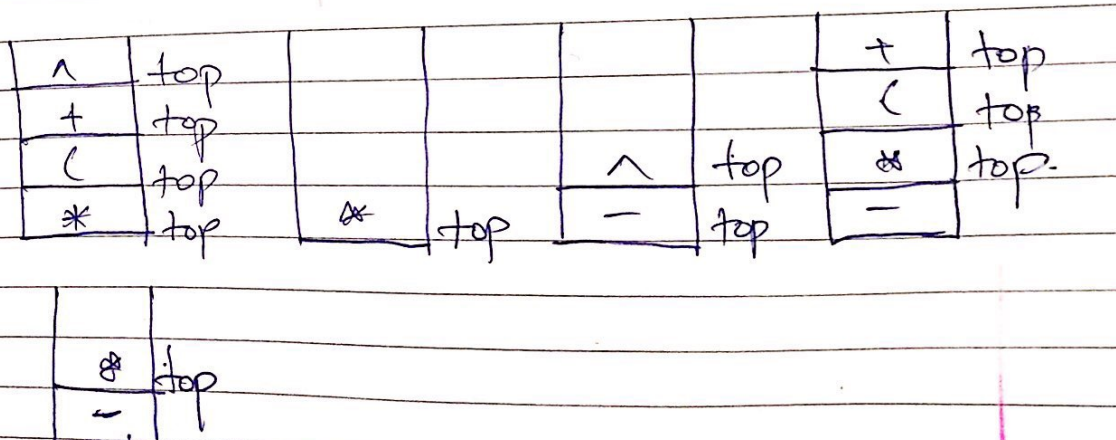
Postfix array: A B C * D / + E F ^ G * -



E.g 2:

Infix array: A | * | (| B | + | C | ^ | D |) | - | E | ^ | F | * | (| G | + | H |)

Postfix array: A B C D ^ + * E F ^ G H + * -

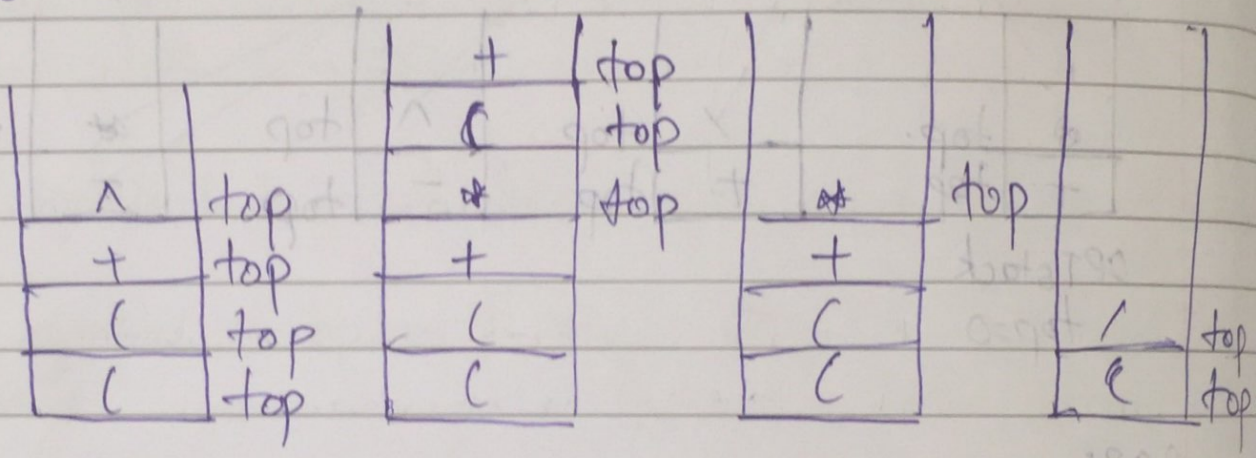


eg3:

((A+B^C*(D+E))/G+H)^I

Infix array: [(A + B ^ C * (D + E)) / G + H) ^ I]

Postfix array: [A B C ^ D E + * + G / H + I ^]



Algorithm for evaluating postfix expression

Step-1 Scan the symbol of postfix array one-by-one from left to right.

Step-2 If symbol is operand then push it to OPND stack.

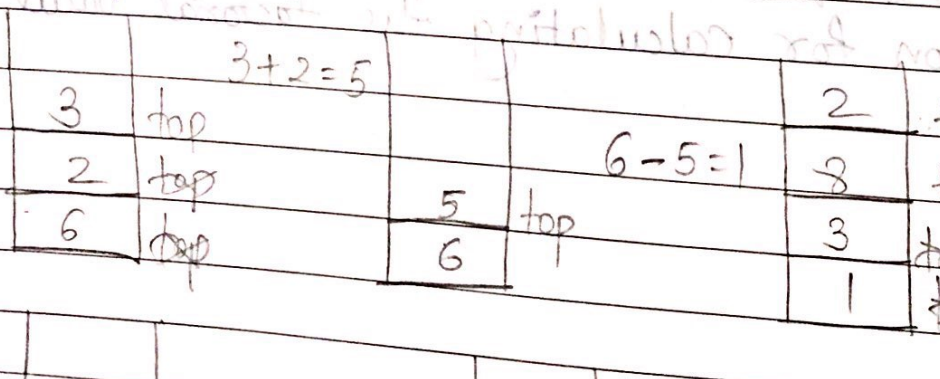
Step-3 If symbol is operator then pop last two elements of OPND stack & evaluate it as [TOP-1] operator [TOP]. & push it to stack.

Step-4 Do the same process until end of postfix array.

Step-5 Pop the element of stack which will be value of evaluation of postfix expression.

E.g:

6	2	3	+	-	3	8	2	/	+	*	2	^	3	+



Next

Recursion:

In C, it is possible for the function to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition' recursion is thus the process of defining something in terms of itself

Let us now see a simple example of recursion. Suppose we want to calculate the factorial value of an integer. As we know, the factorial of a number is the product of all the integers between 1 & that number. For example, 4 factorial is 4*3*2*1. This can also be expressed as 4! = 4*3! where '!' stands for factorial. Thus factorial of a number can be expressed in the form of itself. Hence this can be programmed using recursion. However, before we try to write a recursive function for calculating factorial let us take a look at the non-recursive function for calculating the factorial value of an integer.

```
#include <stdio.h>
int factorial (int); // Function declaration
int main()
{
    int a, fact;

    printf ("Enter any number");
    scanf ("%d", &a);
```

```
    fact = factorial(a); // calling function
    printf("factorial value = %d\n", fact);
    return 0;
}

int factorial(int x) // function definition
{
    int f = 1, i;

    for(i = x; i >= 1; i--)
        f = f * i;

    return(f);
}
```

And here is the output...

```
Enter any number 3
Factorial value = 6
```

Work though the above program carefully, till you understand the logic of the program properly Recursive factorial function can be understood only if you are thorough with the above logic.

Following is the recursive version of the function to calculate the factorial value.

```
#include <stdio.h>
int rec(int);
int main()
{
```

```
    int a, fact;

    printf("Enter any number");
    scanf("%d", &a);

    fact = rec(a);
    printf("Factorial value = %d\n", fact);
    return 0;
}

int rec(int x)
{
    int f;

    if (x==1)
        return(1);
    else
        f = x * rec(x-1);

    return(f);
}
```

And here is the output for four runs of the Program

```
Enter any number 1
Factorial value = 1
Enter any number 2
Factorial value = 2
Enter any number 3
Factorial value = 6
Enter any number 5
```

Factorial value = 120

Let us understand this recursive factorial function thoroughly. In the first run when the number entered through scanf() is 1, let us see what action does rec() take. The value of a (i.e. 1) is copied into x. Since x turns out to be 1, the condition if(x==1) is satisfied & hence 1 (which indeed is the value of 1 factorial) is returned through the return statement.

When the number entered through scanf() is 2, the (x==1) test fails, so we reach the statement,

```
f = x * rec(x-1);
```

And here is where we meet recursion. How do we handle the expression x * rec(x-1)? We multiply x by rec(x-1). Since the current value of x is 2, it is same as saying that we must calculate the value (2 * rec(1)). We knows that the value returned by rec(1) is 1, so the expression reduces to (2*1), or simply 2. Thus, the statement,

```
x * rec(x-1);
```

evaluates to 2, which is stored in the variable f, & is returned to main(), where it is duly printed as,

Factorial value = 2

Now perhaps you can see what would happen if the value of a is 3, 4, 5 & so on.

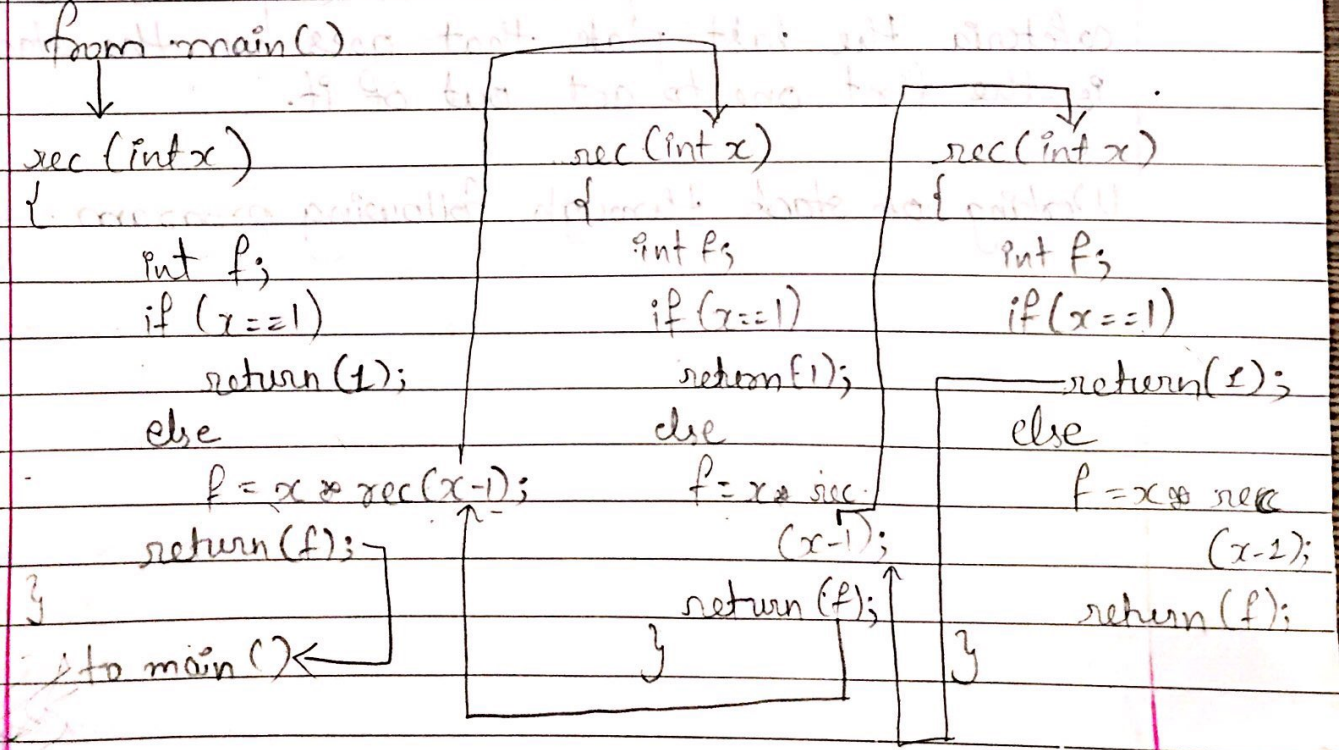
In case the value of a is 5, main() would call rec() with 5 as its actual argument, & rec() will send back the computed value. But before sending the computed value, rec() calls rec() & waits for a value to be returned. It is possible for the rec() that has just been called to call yet another rec(); the argument x being decreased in value by 1 for each of these recursive calls. We speak of this series of calls to rec() as being different invocations of rec(). These successive invocations of the same function are possible because the C compiler keeps track of which invocation calls which. These recursive invocations end finally when the last invocation gets argument value of 1, which the preceeding invocation of rec() now uses to calculate its own f value & so on up the ladder. So,

rec(5) returns (5 times rec(4);
which returns (4 times rec(3),
which returns (3 times rec(2),
which returns (2 times rec(1)
which returns(1)))))

Assume that the no. entered through scanf() is 3. Using figure let's visualize & what exactly happens when the recursive function rec() gets called. The first time when rec()

is called main(), x collects 3. From here, since x is not equal to 1, the if block is skipped & rec() is called again with the argument (x-1), i.e. 2. This is a recursive call. Since x is still not equal to 1, rec() is called yet another time with argument (2-1). This time as x is 1 control goes back to previous rec() with the value 1. & f is evaluated as 2.

Similarly, each rec() evaluates its f from the returned value, & finally 6 is returned to main(). The sequence would be grasped better by following the arrows shown in figure. Let it be clear that while executing the program there do not exist so many copies of the function rec(). These have been shown in the figure to keep a track of the control flows during successive recursive calls.



Recursion may be complicated at first glance, but it is often the most direct way to code an algorithm, & once you are familar with recursion

⇒ Recursion & stack:

There are different ways in which data can be organised. For example, if you are to store 5 numbers then we can store them in five different ways of organizing the data are known as data structures.
The compiler uses one such data structure called stack for implementing normal as well as recursive function calls.
A stack is a last In first Out (LIFO) data structure. This means that the last item to get stored on the stack (often called push operation) is the first one to get out of it (often called POP).
Stack can be compared to the plates in cafeteria the last plate that goes on the stack is the first one to get out of it.

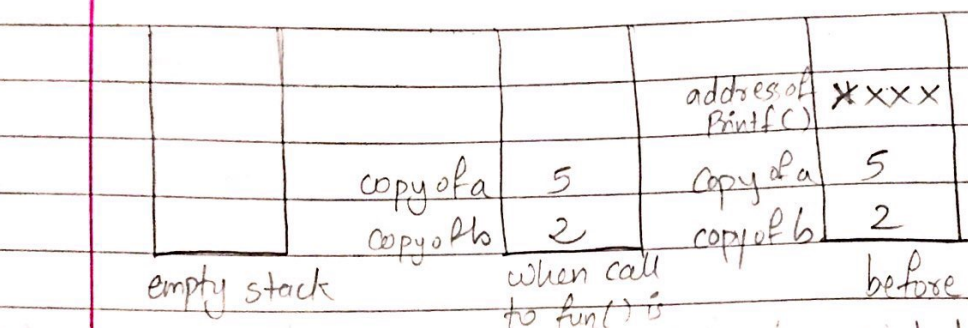
Working of stack through following program.

```
main()
{
    int a=5, b=2, c;
    c=add(a,b);
    printf("Sum=%d",c);
}
add(int i, int j)
{
```

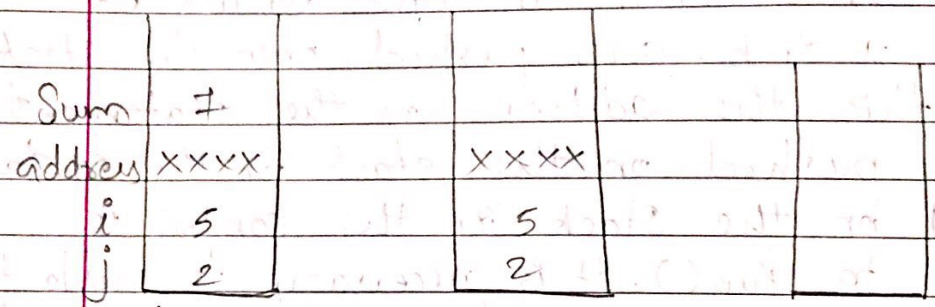
```
    int sum;
    sum = i+j;
    return sum;
}
```

In this program before transferring the execution control to the function fun() the values of parameters a & b are pushed onto the stack. Following this the address of the statement printf() is pushed on the stack & the control is pushed on the stack & the control is transferred to fun(). It is necessary to push this address on the stack. In fun() the values of a & b that were pushed on the stack. When value of sum is returned sum is popped up from the stack. Next the address of the stmt where the control should be returned is popped up from the stack. Using this address the control returns to the printf() statement in main(). Before execution of printf() begins the two integers that were earlier pushed on the stack are now popped off.

How the values are being pushed & popped even though we didn't write any code to do so?
- The compiler on encountering the function call would generate code to push parameters & the address. Similarly, it would generate code to clear the stack when the control returns back from fun(). Figure shows the contents of the stack at different stages of execution.



empty stack | when call to fun() is met | before transferring control to fun()



after control reaches fun() | while returning control from fun() | on returning control from fun().

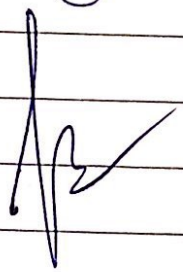
Note that in this program popping of sum & address is done by fun(), whereas popping of the two integers is done by main(). When it is done this way it is known as 'cdecl calling convention'. There are other calling conventions as well where instead of main(), fun() itself clears the two integers. The calling convention also decides whether the parameters being passed to the function are pushed on the stack in left-to-right or right-to-left order. The standard calling convention always uses the right-to-left order. Thus during the call to fun() firstly value of b is pushed to the stack, followed by the value of a.

The recursive calls are no different. Whenever we make a recursive call the parameters

& the return address gets pushed on the stack. The stack gets unwounded when the control returns from the called function. Thus during every recursive function call we are working with a fresh set of parameters.

Also, note that while writing recursive functions you must have an if statement somewhere in the recursive call being executed. If you don't do this & you call the function, you will fall in an indefinite loop, & the stack will keep on getting filled with parameters & the return address each time there is a call.

Soon the stack would become full & you would get a run-time error indicating that the stack has become full. This is a very common error while writing recursive functions.



Menu driven program of stack implementation:

(i) Class definitions:

```
#include<iostream>
using namespace std;
#define size 5 // symbolic constant
class stack_implement
{
        int top, item, freespace;
        int stack[size];
public:
        void push()
        {
                if (top >= size)
                {
                        cout<<" Stack is full\n";
                }
                else
                {
                        top=top+1;
                        cout<<" Enter item=";
                        cin>>item;
                        stack[top]=item;
                }
        }
        void pop()
        {
                if (top==0)
                {
                        cout<<" Stack is empty\n";
                }
```

```
        else
        {
(1) Implementing        item=stack[top];
                        top=top-1;
        }
}
void status()
{
        if(top==0)
        {
                cout<<"stack is empty\n";
        }
        else
        {
                if(top>=size)
                {
                        cout<<"stack is full\n";
                }
                else
                {
                        freespace=size-top;
                        cout<<"Available space in
                                stack is "<<freespace<<"\n";
                }
        }
}
void display()
{
        if(top==0)
        {
                cout<<"stack is empty\n";
        }
```

```
        else
        {
            for (int i=top; i>=0; i--)
            {
                cout << stack[i];
            }
        }
    }
};

int main()
{
    int ch;
    stack_implement S1;
    cout << "Selection \n";
    cout << "1. PUSH \n";
    cout << "2. POP \n";
    cout << "3. STATUS \n";
    cout << "4. DISPLAY \n";
    cout << "Give choice = ";
    cin >> ch;

    switch (ch)
    {
        case 1:
            S1.push();
            break;
        case 2:
            S1.pop();
            break;
        case 3:
            S1.status();
            break;
        case 4:
            S1.display();
            break;
        default:
            cout << "Invalid choice";
    }
}
```

Ch. Queue

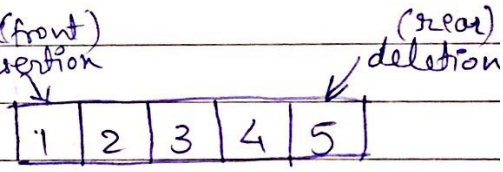
Queue:

- Queue is an order collection of homoge data elements where insertion & deletion operation takes place at two ~~ab~~ extream ends
- It will follow order First In First Out (FIFO)

Types of queue:

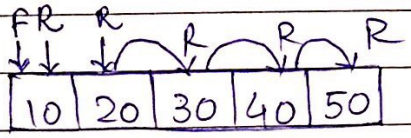
(1) Simple queue
(2) Circular queue
(3) Deque. (~~Dubte~~ Double ended queue)

Insertion
- Enqueue } - Basic operation
deletion
- Dequeue }



again ny insertion
- Front } - Pointers
~~deletion~~ — Rear
Pachar nu

F=R=0



If R = length then Q is full

* Algorithm of enqueue:

Input: An element item that has to be inserted.

output: Inserted item is at the rear of the queue.

Data structure: Array representation of queue.

```
Step-1   If (Rear = length) then
Step-2       print "queue is full"
Step-3       exit
Step-4   else
Step-5       if (Front=0 and rear=0) then
Step-6           front=1
Step-7       end if
Step-8       Rear = Rear+1
Step-9       Q[Rear] = item
Step-10  end if
Step-11  Stop
```

* Algorithm of dequeue:

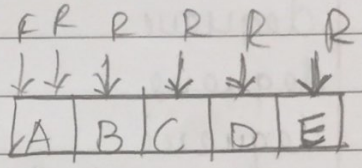
```
Step-1   If (front=0) then
Step-2       print "queue is empty"
Step-3   exit
Step-4   else
Step-5       item = q[front]
Step-6       if (front=rear) then
Step-7           front=rear=0
Step-8       else
Step-9           front = front+1
Step-10      end if
```

front: front and rear

Step-11 end if
Step-12 stop

E.g:
Assume that length of the simple queue is five perform following operation of queue on that

enqueue (A)
(B)
(C)
(D)
(E)

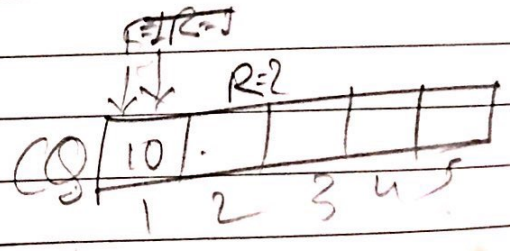


dequeue
dequeue
dequeue (F)
dequeue
dequeue
dequeue

E.g:
Assume the length of the Q is 6 & perform the following operation on queue.

front = 3
rear = 4

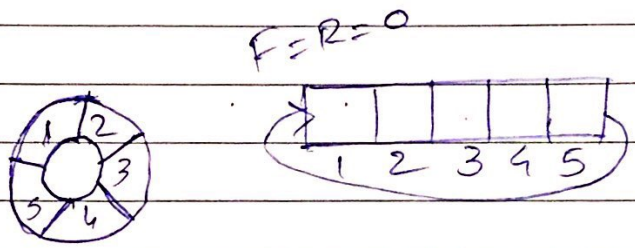
dequeue
dequeue
dequeue
dequeue (10)
enqueue (20)
enqueue (30)
enqueue (40)
dequeue
enqueue (50)
enqueue (60)
enqueue (70)
dequeue



next = 1 % 5 + 1
= 1 + 1
= 2

Circular queue:

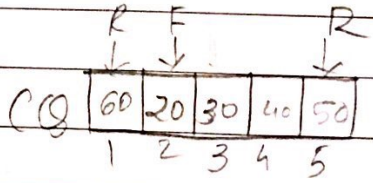
- It is indicated by CQ.



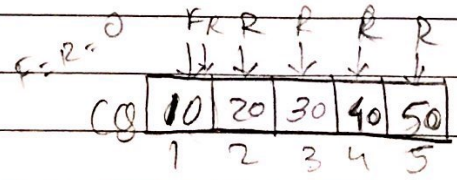
Algorithm of Encqueue:

Step-1 If (Front = 0) then
Step-2 Front = 1
Step-3 Rear = 1
Step-4 CQ[Rear] = item
Step-5 Else
Step-6 next = Rear mod length + 1 (%)
Step-7 IF (next ≠ Front) then
Step-8 Rear = next
Step-9 CQ[Rear] = item
Step-10 else
Step-11 Print "queue is full"
Step-12 End if
Step-13 End if
Step-14 stop

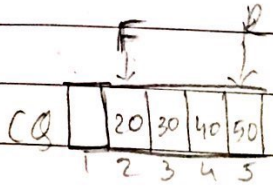
next = 1 % 5 + 1
= 1 + 1
= 2



next = 1 % 5 + 1
= 1 + 1
= 2



next = 2 % 5 + 1
= 2 + 1
= 3



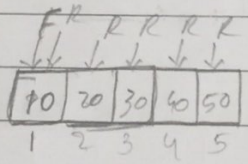
next = 3 % 5 + 1
= 3 + 1
= 4

next = 5 % 5 + 1
= 0 + 1
= 1

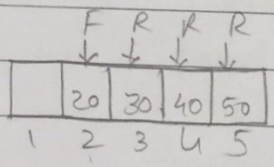
next = 4 % 5 + 1
= 4 + 1
= 5

Algorithm of Deletion queue:

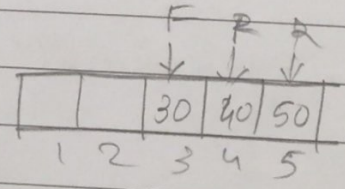
Step-1 If (front=0) then
Step-2 print "queue is empty"
Step-3 exit
Step-4 else
Step-5 Item = Q[front]
Step-6 if (front = rear) then
Step-7 front = rear = 0
Step-8 else
Step-9 front = front (%) mod length + 1
Step-10 endif
Step-11 end if
Step-12 Stop



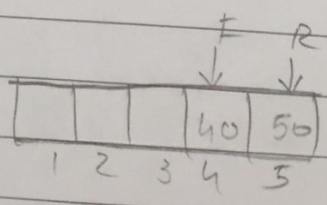
10 = Q[1]
front = 1 % 5 + 1
= 1 + 1
= 2



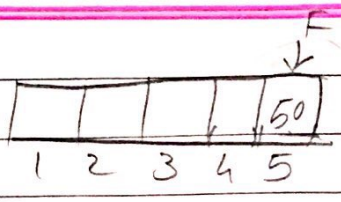
front = 2 % 5 + 1
= 2 + 1
= 3



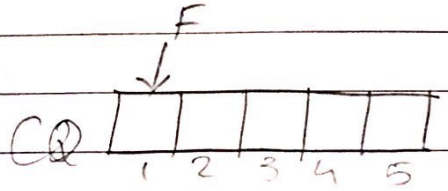
front = 3 % 5 + 1
= 3 + 1
= 4



front = 4 % 5 + 1
= 4 + 1
= 5

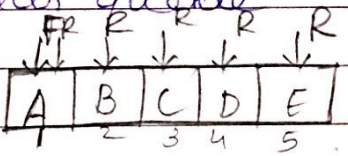


front = 5 % 5 + 1
= 0 + 1
= 1

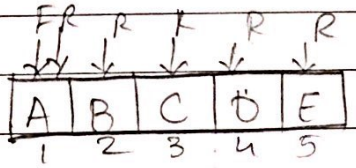


* E.g:

Assume the queue size is 5 & perform following operations on circular queue.



Encqueue (A)
(B)
(C)
(D)
(E)
(F)

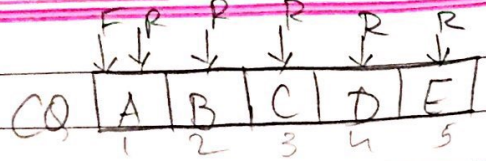


→ queue is full

Decqueue
Encqueue (F)
(G)

Decqueue
Decqueue
Decqueue
Decqueue
Decqueue

Give final status of queue.

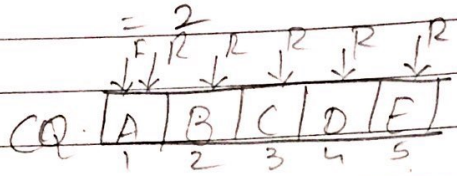


next = 1 % 5 + 1
= 1 + 1
= 2

next = 2 % 5 + 1
= 2 + 1
= 3

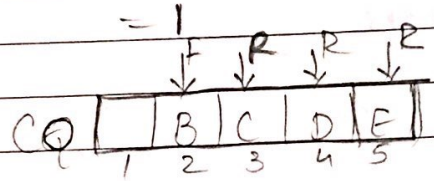
next = 3 % 5 + 1
= 3 + 1
= 4

next = 4 %
= 4
= 5

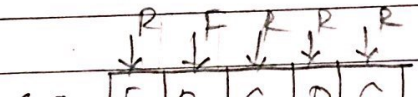


→ queue is full

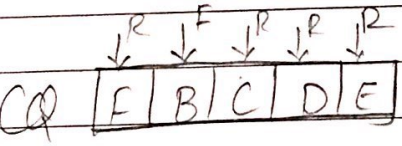
next = 5 % 5 + 1
= 0 + 1
= 1



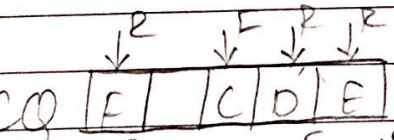
front = 1 % 5 + 1
= 1 + 1
= 2



next = 5 % 5 + 1
= 0 + 1
= 1



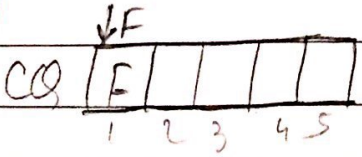
→ queue is full



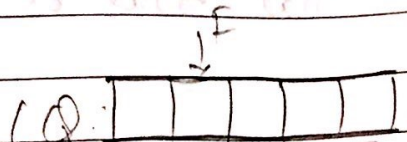
front = 2 % 5 + 1
= 2 + 1
= 3

front = 3 % 5 + 1
= 3 + 1
= 4

front = 4 % 5 + 1
= 4 + 1
= 5



front = 5 % 5 + 1
= 0 + 1
= 1



queue is em

Simple queue:

```
#include<iostream>
using namespace std;
#define length 5

class simple_queue
{
        int front, rear, item;
        int queue[length];

    public:
            simple_queue()
            {
                front = 0;
                rear = 0;
            }
            void enqueue()
            {
                if (rear==length)
                {
                    cout<<"queue is full \n";
                }
                else
                {
                    if (front==0 && rear==0)
                    {
                        front=1;
                    }
```

```
            queue[rear] = item;
        }
    }
    void dequeue()
    {
        if (front == 0)
        {
            cout << "queue is empty \n";
        }
        else
        {
            item = queue[front];
            if (front == rear)
            {
                front = 0;
                rear = 0;
            }
            else
            {
                front = front + 1;
            }
        }
    }
    void display()
    {
        if (front == 0)
        {
            cout << "queue is empty \n";
        }
        else
        {
            for(int i = front; i <= rear; i++)
```

```
        {
            cout << queue[i];
        }
    }
};

int main()
{
    int ch;

    simple_queue SI;

    while (1)
    {
        cout << "Select \n";
        cout << "1. Enqueue\n";
        cout << "2. Dequeue\n";
        cout << "3. Display\n";
        cout << "4. Exit\n";
        cout << "Give choice=";
        cin >> ch;

        if (ch == 4)
        {
            cout << "Thank You\n";
            break;
        }
        else
        {
            switch (ch)
            {
                case 1:
```

```
            st.enqueue();

        case 2:
            st.dequeue();

        case 3:
            st.display();

        default:
            cout<<"Invalid choice\n";

        }
    }
}
    return 0;
}
```

Pointer:

→ * - value at address operator.

int i = 3; ← Reserve the space
Associate Name
Store the value.

e.g:

```
int main()
{
    int i=3;
    int *j;
    int **k;
    j = &i;

    printf("Address i= %u\n", &i);
           ("   "  "  =%u\n", j);
           ("   "  "  =%u\n", &j);
           (j);
           (u);
          *(&i) (*j);
}
```

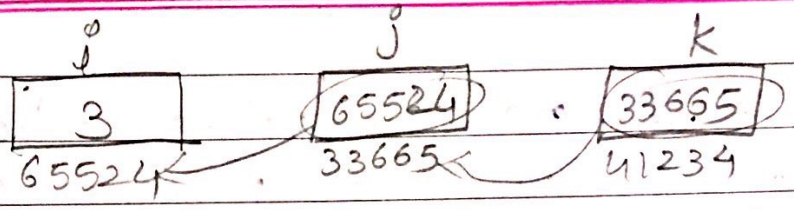
i = 3
j = &i
k = &j
if printf (**k)

int **k; ← pointer variable address will store

→ output = 65524 → &i
65524 → j
33665 → &j
65524 → j
3 → i
3 → *(&i)
3 → (*j)

i = *j . is same

same so it will print same



print(**k) → 3

→ If we want to access a variable from class pointer then instead of (.) dot we will used (→) arrow.

e.g:

```
class test
{
    public:
        int x;
        int *y;
};

int main()
{
    test t;
    t.x=10;
    t->y=&t.x;
}
```

→ If object of class is pointer then the member of class variable will be access using (→) arrow operator.

e.g:

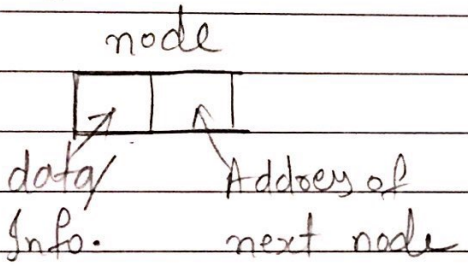
```
class test
{
    public:
        int x;
        int *y;
};

int main()
{
    test *t;
    t->x;
    t->y;
}
```

Ch-3: Linked List

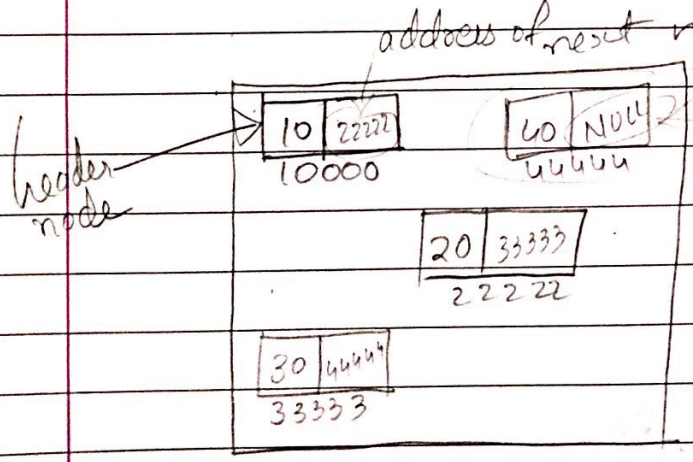
* Linked list:

element of linked list

- A linked list is an ordered collection of finite, homogenous data element called "nodes" where the liner order is maintain by means of links or pointers.



→ First node is known as header node.
→ ~~Address of first node is only~~.
→ We have only address of first node.
→ The node which have next part "NULL" then it is a last node.



indicate NULL

- class node
{
    int data;
    node * next; ← It will store address of next node
    node (int info, node * n = NULL) ← If we want next part of node as NULL so we have initialized it as a NULL
    {
        data = info;
        next = n;
    }
};

class name

- whenever we create new node at that time one part is of data & another will be NULL.

class list

→ Operations allowed on linked list:
- Creation of linked list.
(1) Insertion ← First, Middle, Last
(2) Deletion ← F, M, L
(3) Traversal ← F, M, L
(4) Searching ← F, M
(5) Merging ← F, L, M, L
(6) Copying ← F, M, L

node *head, *new

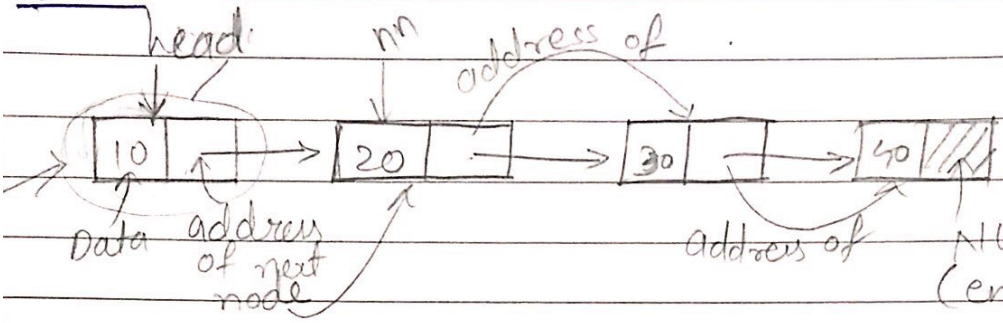
- Types of linked list:

(1) SLL → Singly linked list
(2) SCLL → Singly circular linked list
(3) DLL → Doubly linked list
(4) DCLL → Doubly circular linked list

object create → student S

S=new student();

SLL:



- Initial value of head is NULL which means linked list is empty.

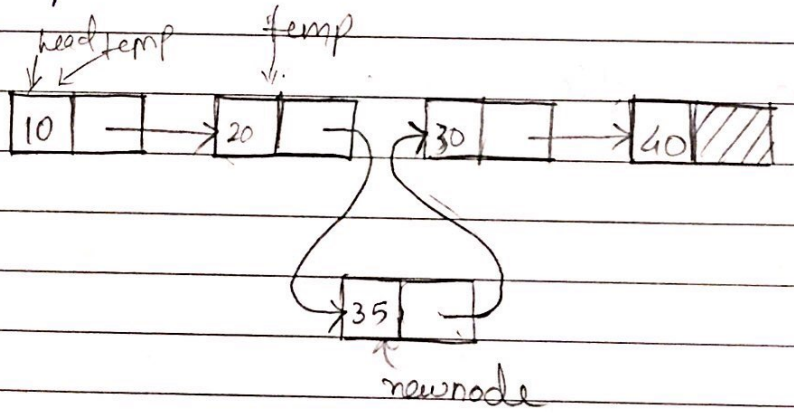
Creation:

```
do
{
    if head=NULL then
        head=newnode=new node(data);
    else
        newnode→next=new node(data);
        newnode=newnode→next;
    endif
    take choice from user
} while(ch='Y');
```

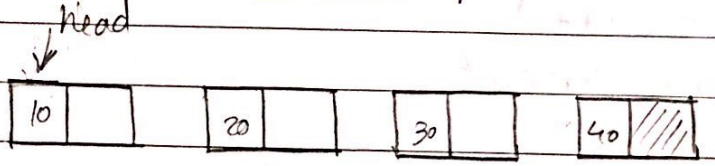
newnode & head are pointer variables

* Insertion at middle position:

Step-1 temp = head
Step-2 newnode = new node (data)
Step-3 take Pos from user
Step-4 for (i = 1; i < Pos-1; i++)
Step-5 temp = temp → next
Step-6 end for
Step-7 newnode → next = temp → next
Step-8 temp → next = newnode
Step-9 Stop.

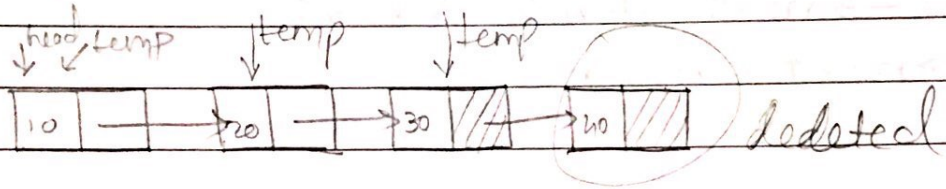


* Deletion from 1st position:

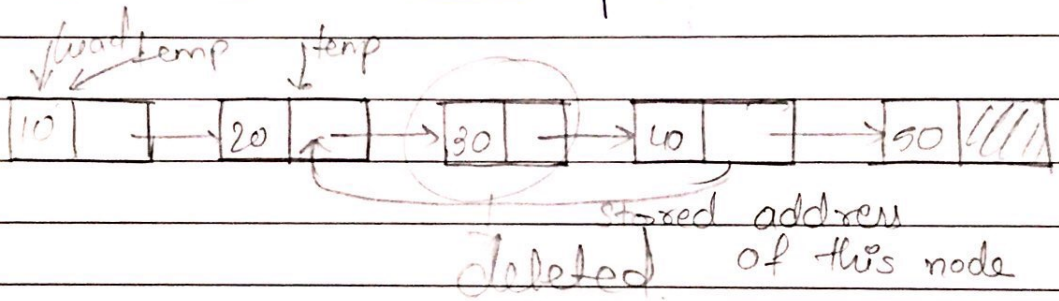


Step-1 head = head → next

Step-1 temp = head
Step-2 while (temp → next → next != NULL) do
Step-3 temp = temp → next
Step-4 end while
Step-5 temp → next = NULL
Step-6. Stop



Deletion from middle position:

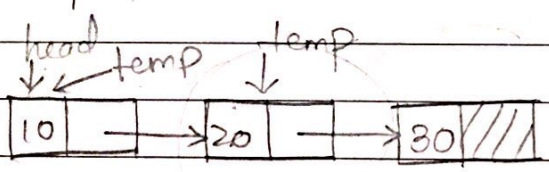


Step-1 temp = head
Step-2 for (i=1; i<pos-1; i++)
Step-3 temp = temp → next
Step-4 end for
Step-5 temp → next = temp → next → next
Step-6 Stop

Searching from linked list:

Step-1 temp = head, found = 0

Step-2 while (temp -> next != NULL and found = 0) do
if compare (temp -> data, key) true then
found = 1
else
temp = temp -> next
end if
end while
if (found = 0) then
print "key not found"
else
print "key found"
end if
Stop



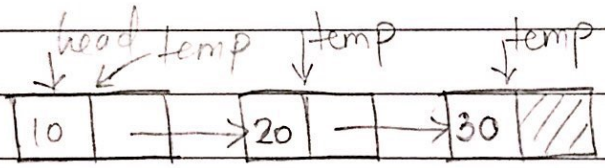
key = 20

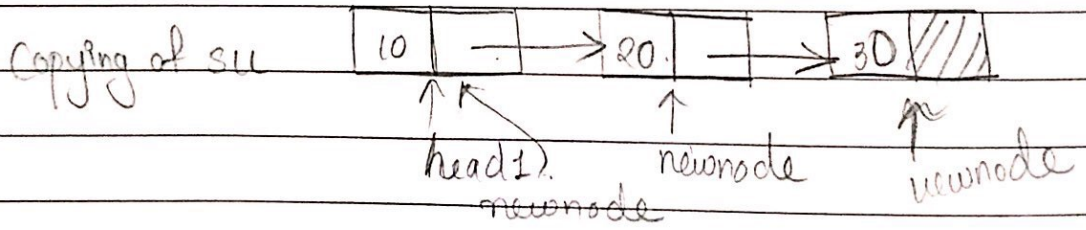
found = 1
key found.

temp != NULL ← last node par temp pointer jayse liyare.

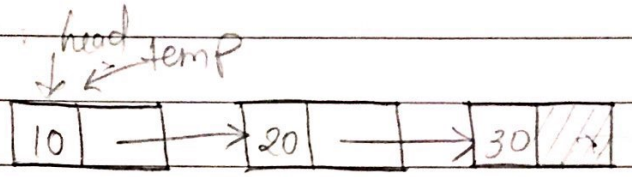
Copying of SLL:

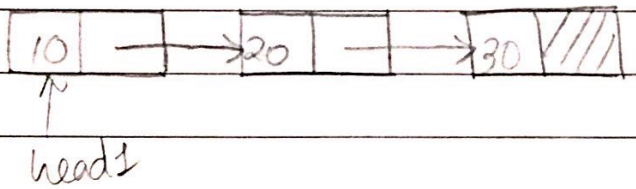
```
temp = head
while temp != NULL do
    if head1 = NULL then
        head1 = newnode = new node(temp -> data)
    else
        newnode -> next = new node(temp -> data)
        newnode = newnode -> next
    end if
    temp = temp -> next
end while
stop
```





Merging of two SLL:





Step-1 temp = head.

while temp→next = NULL do

temp→next = head1.

- ~~NANR~~

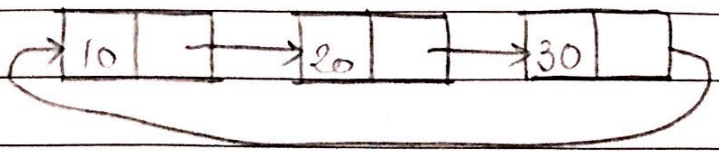
- ~~WANR~~

- Functions:

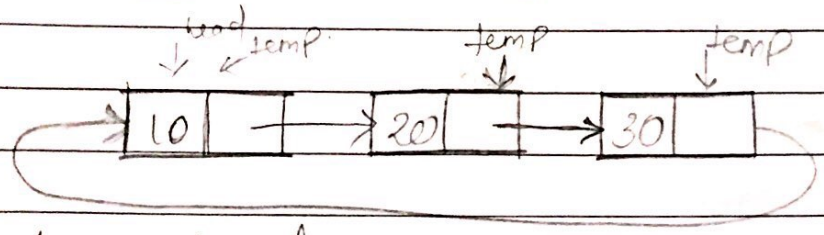
- NANR (
  WANR (
  WAWR (
  NAWR (

SCLL creation:

Step-1 do
Step-2 {
Step-3 if head = NULL then
head = newnode = new node (data);
else
newnode -> next = new node (data);
newnode = newnode -> next
endif
newnode -> next = head.
take choice from user
} while (ch = 'y');

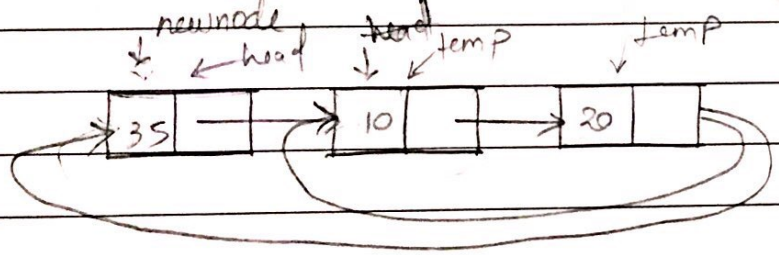


Traversal of SCLL:



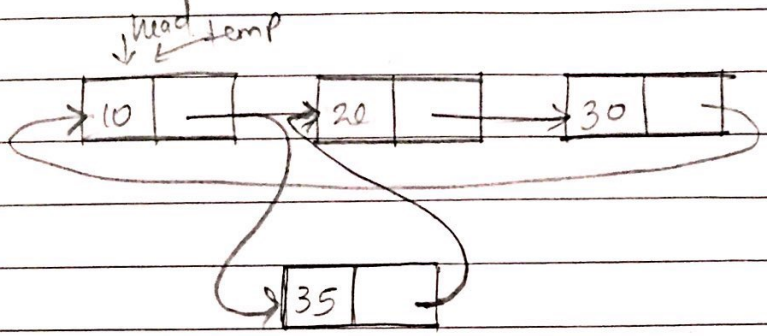
Step-1 temp = head
Step-2 do
Step-3 {
Step-4 Process (temp → data)
Step-5 temp = temp → next
Step-6 } while (temp != head)

Insertion at 1st position:



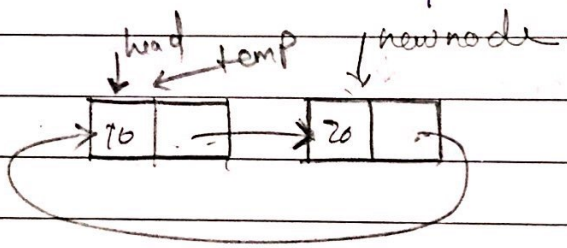
Step-1 temp = head
newnode = new node (data)
while (temp → next != head) do
temp = temp → next
end while
newnode → next = head
temp → next = newnode
head = newnode.

* Insertion at middle position:



→ Same as insertion at middle position of SLL.

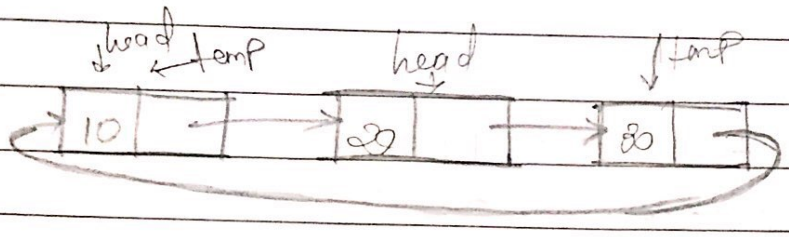
* Insertion at last position:



Step-1

```
temp = head
newnode = new node (data)
while (temp → next != head) do
        temp = temp → next
end while
newnode → next = head
temp → next = newnode
```

Deletion from 1st Position:

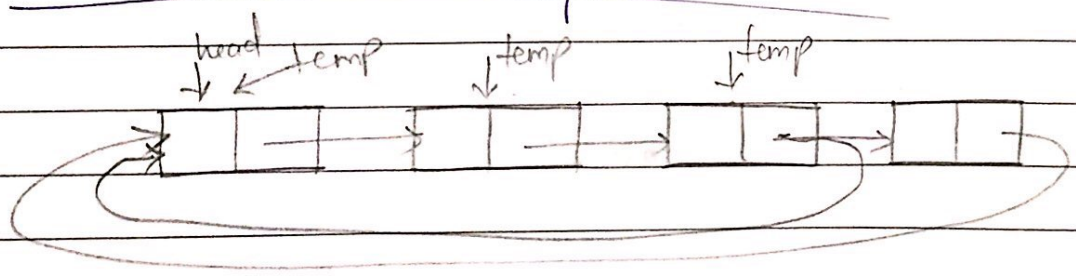



```
temp = head
while (temp -> next != head) do
        temp = temp -> next
end while
head = head -> next
temp -> next = head
```

Deletion from middle position:

→ Same as 0&L middle position deletion

Deletion from last position:

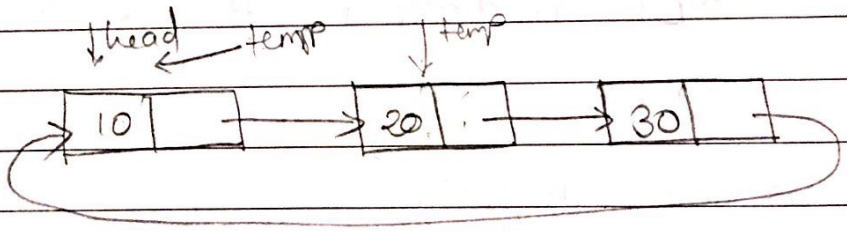



```
temp = head
while (temp -> next -> next != head)
        temp = temp -> next
end while
temp -> next = head
Stop
```

H.W

Searching from SLL:

Step-1 temp = head, found = 0
while (temp -> next != head and found = 0) do
if compare (temp -> data, key) true then
found = 1
else
temp = temp -> next
end if
end while
if (found = 0) then
print "key not found"
else
print "key found"
end if
Stop.

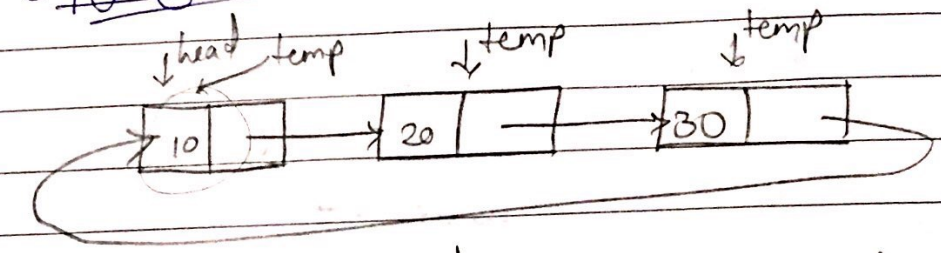


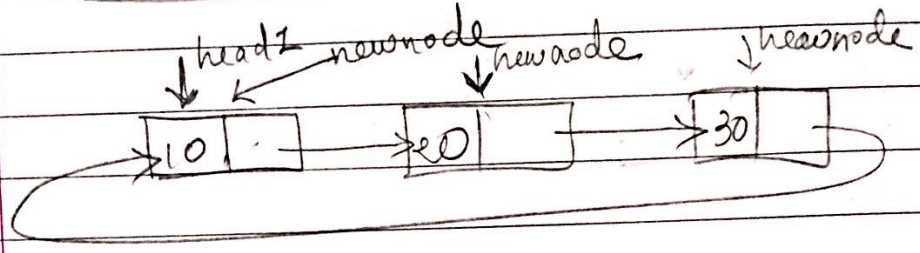
key = 20
found = 1

Step-1 temp = head, found = 0
take a key from user
do
{
If Compare (temp → data, key) = true then
found = 1
else
temp = temp → next
end if
} while (temp != head and found = 0)

If (found = 0) then
print " Not found"
else
print " found"
end if

Copying of SCLL:

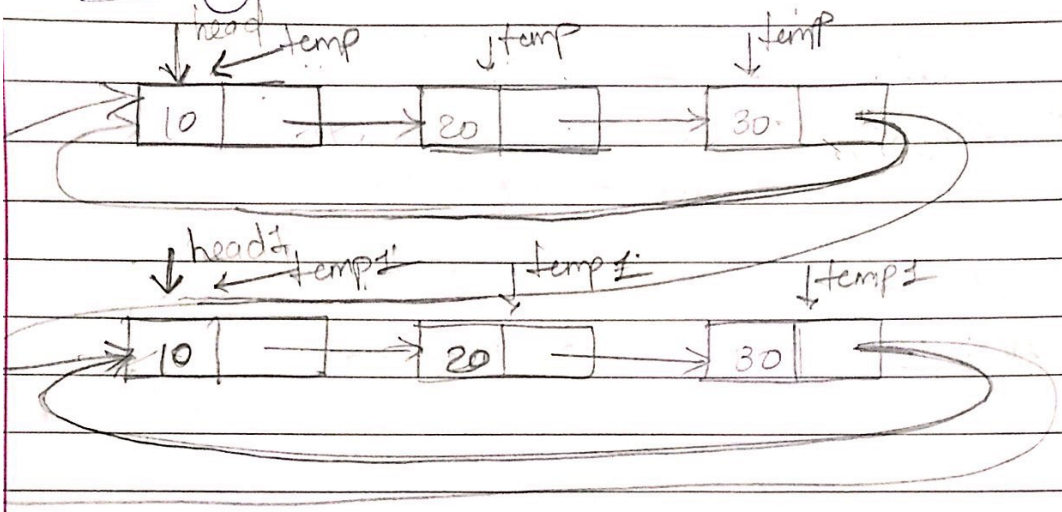




Step-1 temp = head

```
do
{
    if (head1 = NULL) then
        head1 = newnode = new node (temp->data)
    else
        newnode -> next = new node (temp->data)
        newnode = newnode -> next
    end if
        temp = temp -> next
} while (temp != head)
    newnode -> next = head1
Stop
```

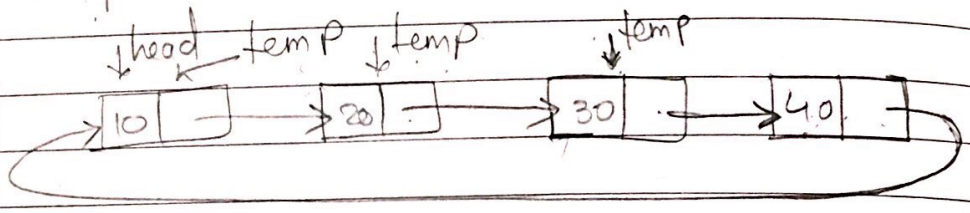
Merging of SCLL:




```
temp = head
while (temp → next != head) do
    temp = temp → next
end while
while (temp → next = head) do
        temp → next = head1
end while
temp1 = head1
while (temp1 → next != head1) do
        temp1 = temp1 → next
end while
temp1 → next = head
```

Q) Algo of insertion at last position:

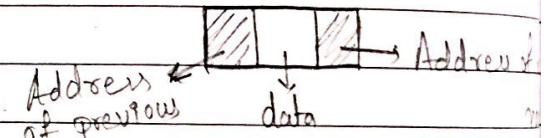
Step-1
temp = head
while (temp -> next != head) do
    temp = temp -> next
end while
temp -> next = new node (data, head);



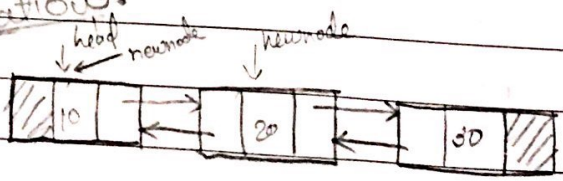
* DCL (Doubly Linked List):

=> class node
{
    int data;
    node * next;
    node * pervious;

    node (int info, node *p = NULL, node *n = NULL)
    {
        data = info;
        previous = p;
        next = n;
    }
};



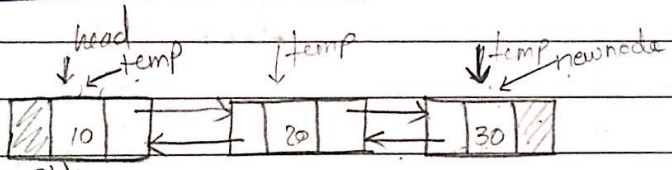
* Creation:



```
do
{
    if head = NULL then
        head = newnode = new node(data);
    else
        newnode -> next = new node (data);
        newnode -> next -> previous = newnode;
        newnode = newnode -> next
    end if
    take choice from user
} while (ch = 'Y')
```

Traversal of DLL:

Traversal of DLL are same SLL



- In Traversal of DLL left to right Traversal of SLL are same
- In DLL right to left is possible

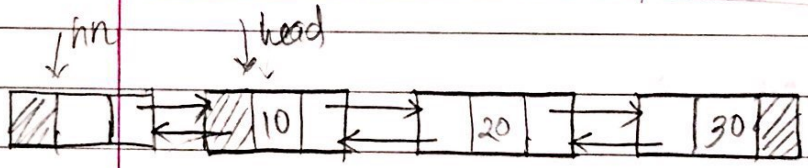
```
temp = head
while (temp != NULL) do
        process (temp -> data)
        temp = temp -> next.
end while
```

Reverse of Traversal:

```
temp
while (temp
        process (temp -> data)
        temp = temp -> previous
end while
```

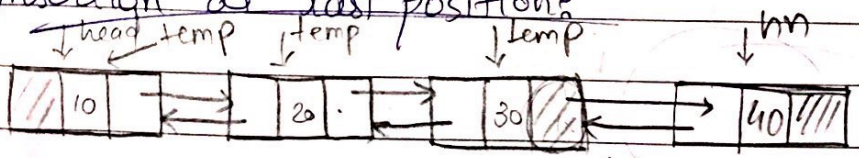
```
temp = head
while (temp → next != NULL) do
        temp = temp → next
end while
while (temp != NULL) do
        process (temp → info)
        temp = temp → previous
end while
```

→ Insertion at first position in DLL:



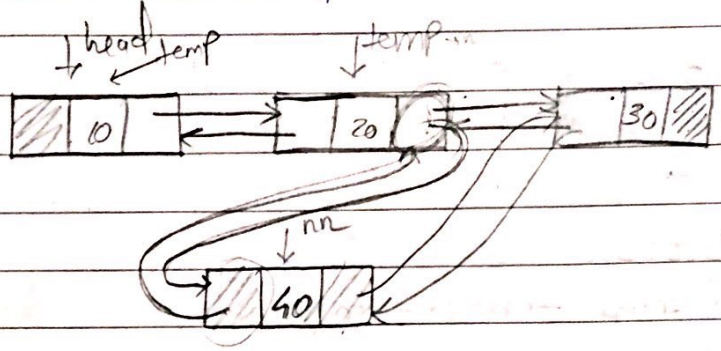
Step-1
```
newnode = new node(data)
newnode → next = head
head → previous = newnode
head = head → previous
```

→ Insertion at last position:




```
temp = head
while (temp → next != NULL) do
        temp = temp → next
end while
newnode = new node (data)
temp → next = newnode
newnode → previous = temp
```

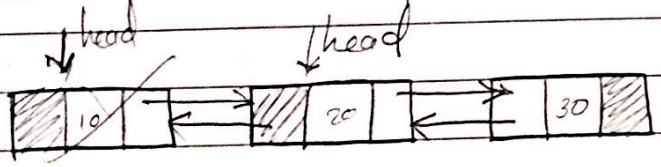
Insertion at Middle position:



pos = 3

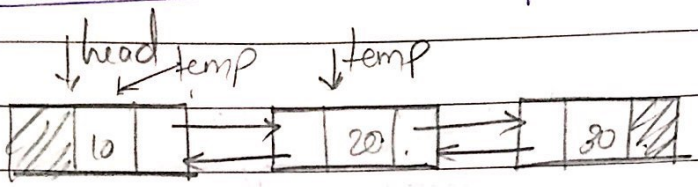
```
temp = head
newnode = new node (data)
take pos from user
for (i = 1; i < pos-1; i++)
        temp = temp → next
end for
newnod → next = temp → next
temp → next → previous = newnode
temp → next = newnode
newnode → previous = temp
```

Deletion from 1st position:



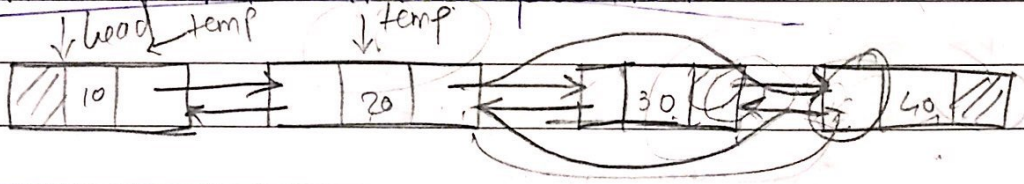
Step-1
```
head = head → next
head → previous = NULL
```

Deletion from last position:



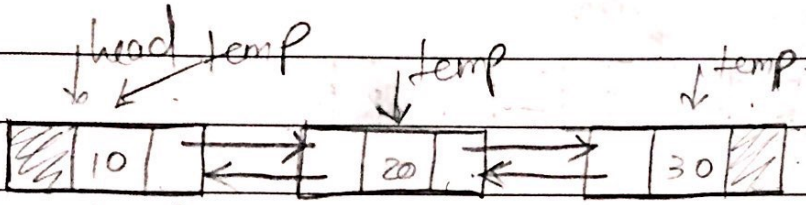

```
temp = head
while (temp → next → next != NULL) do
        temp = temp → next
end while
  temp → next = NULL
stop
```

Deletion from middle position:




```
temp = head
take pos from user
for (i=1; i<pos-1; i++)
        temp = temp → next
end for
temp → next = temp → next → next
temp → next → previous = temp
stop
```

Searching from DLL:

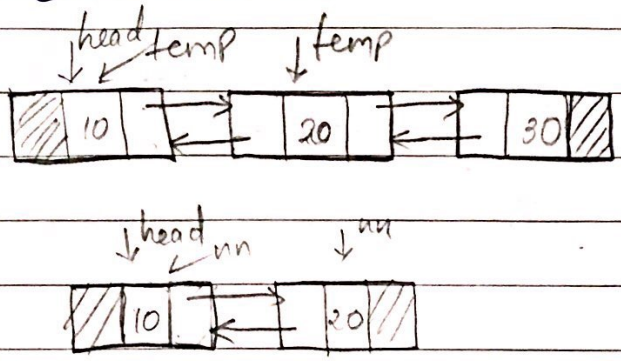


temp = head, found = 0 key = 30

take key from user

```
while (temp != NULL and found = 0)
        if compare (temp -> data, key)
                found = 1
        else
                temp = temp -> next.
        endif
end while
if (found = 0) then
        print "Not found"
else
        print "found"
endif
```

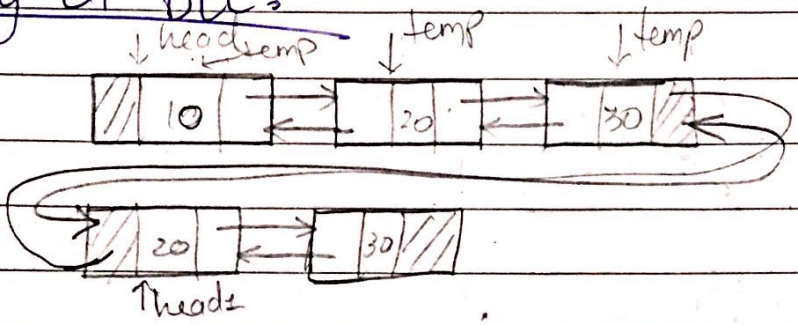
→ Copying of DLL:



Step-1

```
temp = head
while (temp != NULL) do
    if (head1 = NULL) then
        head1 = newnode = new node (temp->data)
    else
        newnode -> next = new node (temp->data)
        newnode -> next -> previous = newnode
        newnode = newnode -> next
    end if
    temp = temp->next
end while
```

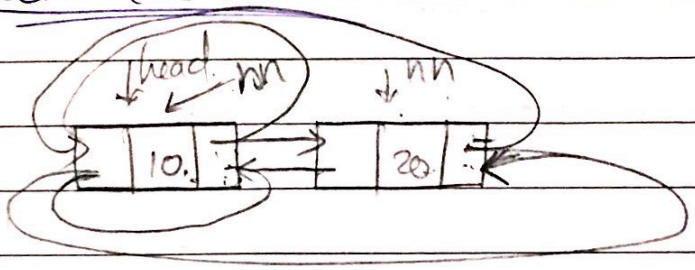
Merging of DLL:



temp = head
while (temp → next != NULL) do
    temp = temp → next
end while
temp → next = head1
head1 → previous = temp
stop

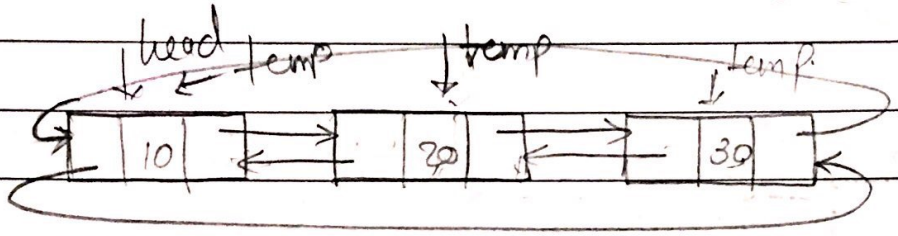
DCLL:

→ Creation of DCLL:



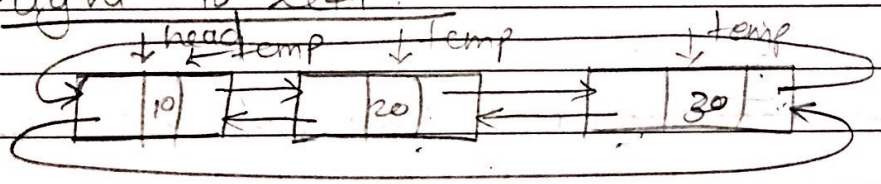

```
do
{
    if (head=NULL) then
        head = newnode = new node (data)
    else
        newnode→next = new node (data)
        newnode → next → previous = newnode
        newnode = newnode → next
    end if
    newnode → next = head
    head → previous = newnode
    take choice from user
} while (ch='Y')
```

→ Traversal of DCLL: left to right

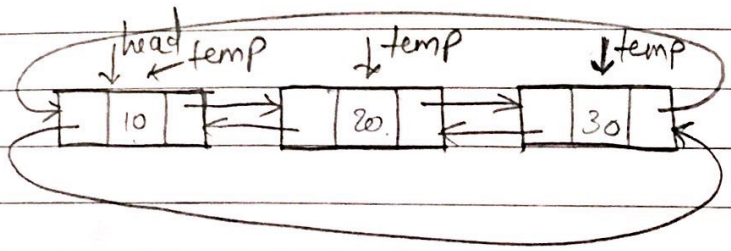



```
temp = head
do
{
        process(temp->data)
        temp = temp -> next
} while (temp != head)
```

→ right to left:



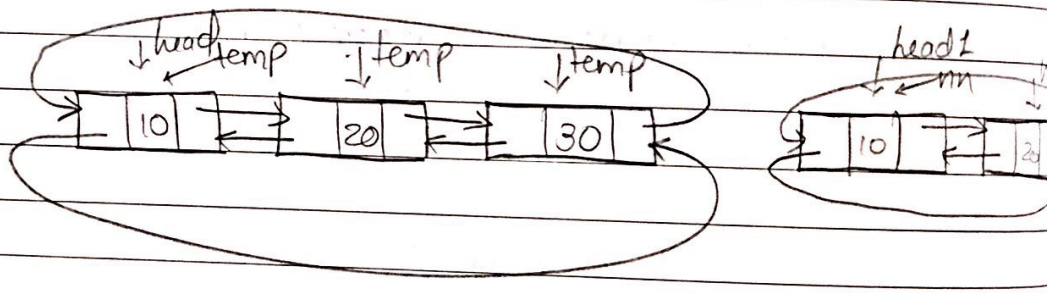
→ Searching of DCLL:



key = 30
found = 0
1
temp->data:

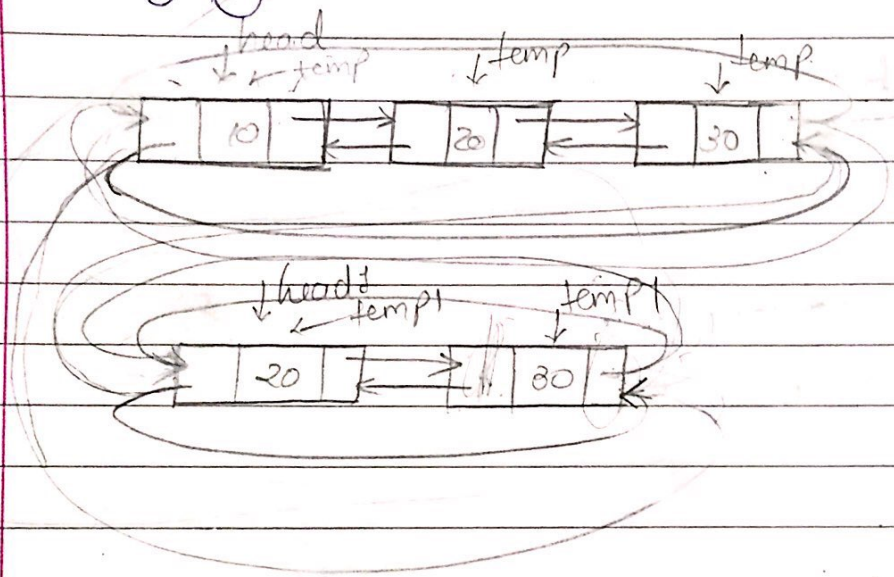
~~ten~~ Same as SCLL

→ Copying of DCLL:



temp = head
do
{
if head1 = NULL do
head1 = newnode = new node (temp→data)
else
newnode→next = new node (temp→data)
newnode→next→previous = newnode
newnode = newnode→next
end if
newnode→next = head1
head1→previous = newnode
temp = temp→next
}while (temp != head)

Merging of DCLL:

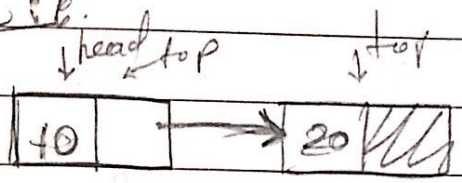



```
temp = head
while (temp → next != head) do
        temp = temp → next
end while
while (temp → next = head) do
        temp → next = head1

end while
temp1 = head1
while (temp1 → next != head1) do
        temp1 = temp1 → next
end while
temp1 → next = head
head → previous = temp1
```

Linked stack & Linked Queue:

Linked stack:

→ Push:

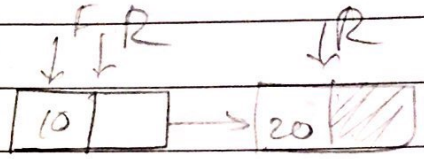



```
if (head = NULL) then
    head = top = new node (data)
else
    top → next = new node (data)
    top = top → next
end if
```

→ POP:

```
temp = head                    if (top = NULL) then
                               Print "deletion not Possible".
                          else
while (temp → next != top) do
    temp = temp → next
end while
top = temp
temp → next = NULL
```

→ Linked Queue:

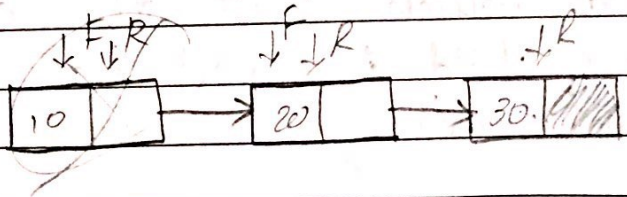


→ Enqueue:

```
if (front = NULL) then
        front = rear = new node (data)
else
        rear → next = new node (data)
        rear = rear → next
end if
```

→ dequeue:

```
if (front = NULL & rear = NULL) then
        print "deletion not possible"
else
        if (front = rear) then
                front = rear = NULL
        else
                front = front → next
        end if
end if
```



Tree

→ **Node:**
- This is a main component of any tree structure. A node of a tree stores the actual data & links to the other node

→ **Parent:**
- The parent of a node is the immidiate pre-dessecer of a node.

→ **Child:**
- Immidiate successers of a node are known as child.

→ **Link:**
- This is a pointer to a node in a tree & RC are two links of a node.

→ **Root:**
- This is a specially designated node which has no parent.

→ **Leaf:**
- The node which is at the end and does not have any child is called leaf node (Terminal node) (External node).

→ **Level:**
- Level is the rank in the hierarchy the root node has level 0.

→ **height:**

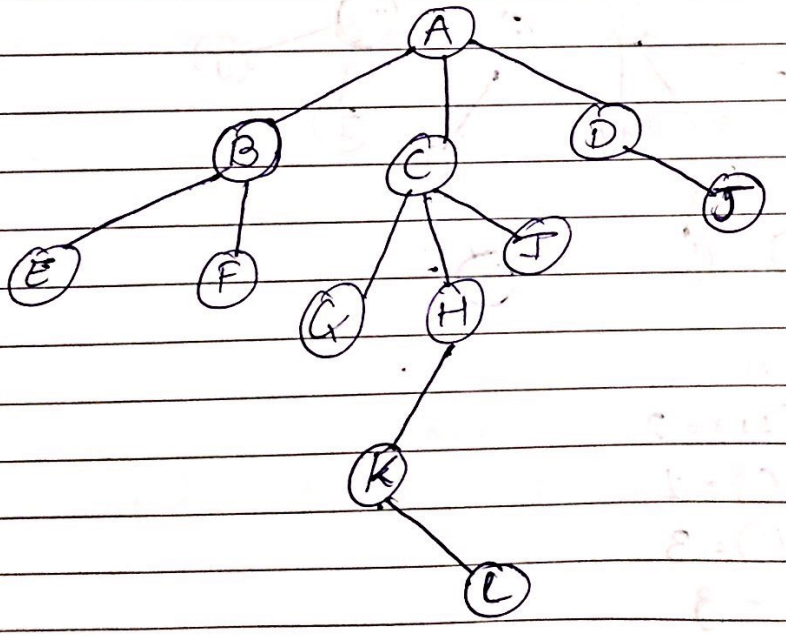
- The maximum number of nodes that is possible in a path starting from the root node to a leaf node is called the height of a tree.

→ **Degree:**

- The maximum number of children that is possible for a node is known as the degree of a node.

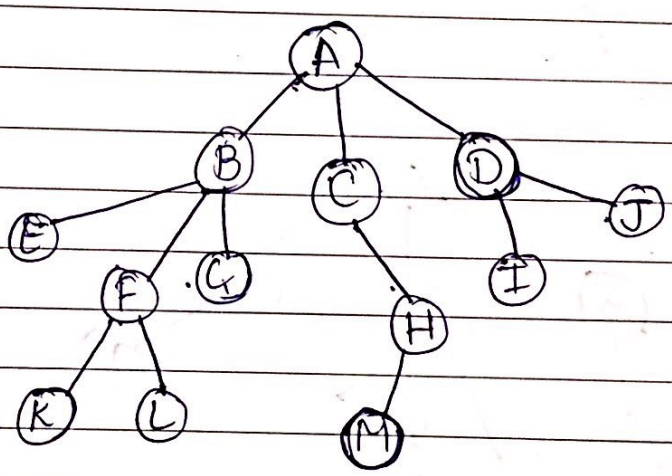
→ **sibling:**

- The nodes which have the same parent are called siblings.



Q: Observe the given tree and find the following with reference to

① The height of tree
② Level of H
③ Level of C
④ Level of K
⑤ Degree of the tree
⑥ The longest path in a tree
⑦ Parent of m
⑧ Sibling (I)
⑨ Child (B)



→① Height - 4
→② Level (H) = 2
Level (C) = 1
Level (K) = 3
→③ Degree - 3
→④ ① A-B-F-L.
② A-C-H-M
→⑤ Parent (M) = H
→⑥ Sibling (I) = J

* Binary tree:

- A binary tree is a special form of a tree. Binary tree can be define as a finite set of nodes such that,

-(i) - The Tree contains a specially designated node called the root of T and the remaining nodes of T form to disjoin binary tree's T_1 and T_2 which are called the left sub tree and the right sub tree.

* Properties of binary tree:

Lemma 1:

- In any binary tree the maximum number of nodes on level L is 2^L where $L \geq 0$.

Lemma 2:

- The maximum number of nodes possible in a binary tree of height h is $2^h - 1$.

Lemma 3:

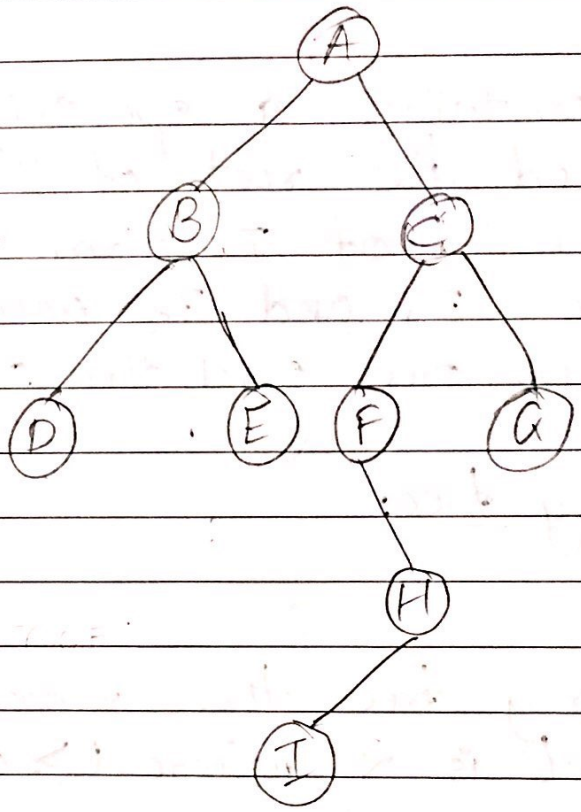
- The minimum number of nodes possible in a binary tree of height h is h.

Lemma 4:

- For any non-empty binary tree if n is the number of nodes & e is the number of edges then $n = e + 1$

Lemma 5:

- For any non-empty binary tree if n_0 is the number of leaf nodes ($deg=0$) and n_2 is number of internal nodes ($deg=2$) then $n_0 = n_2 + 1$



Lemma 6:

- In a link

Representation of binary tree:

1. Array
2. Linear
3. Link → dynamic representation.

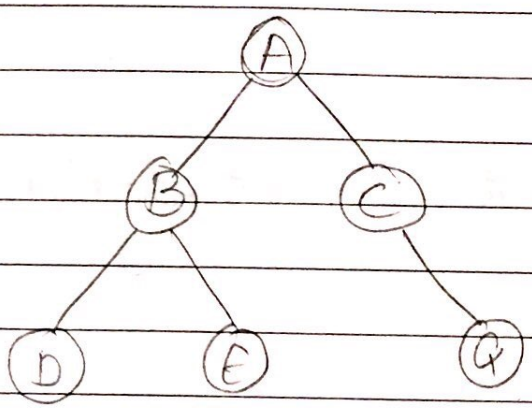
→ (1) Array:

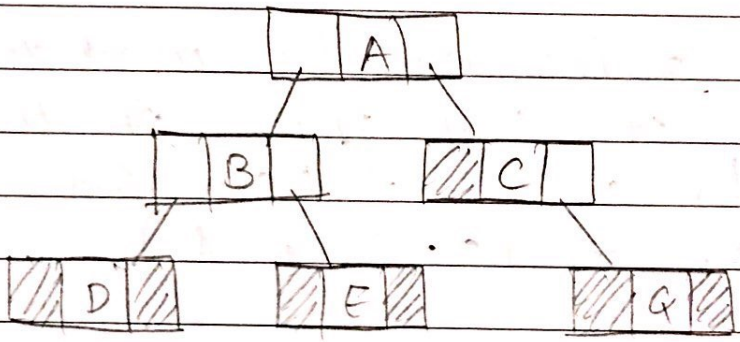
- Following rules can be used to decide the location of any node ~~in the~~ of a tree in the array. (Assuming that the array index starts from 1).

Rule 1:
- The root node is at location 1.

Rule 2:
- For any node with index i, $1 < i \leq n$:
  - (a) Parent (i) = $i/2$
  - (b) L child (i) = $2 \times i$
  - (c) R child (i) = $2 \times i + 1$

→ (2) Link representation:





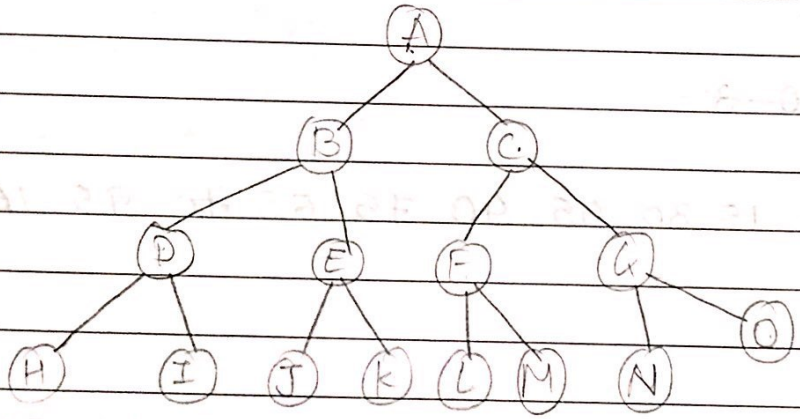
- Lemma1:

- In a link representation of a bina
tree, if there are n number of nodes the
the number of null links $\lambda = n+1$.

Operation on binary tree:

1. Inorder (L, R, r) → left, Root, right
2. Preorder (R, L, r)
3. Postorder (L, r, R)

→ 1. Insertion
2. Deletion
3. Traversal



Inorder:

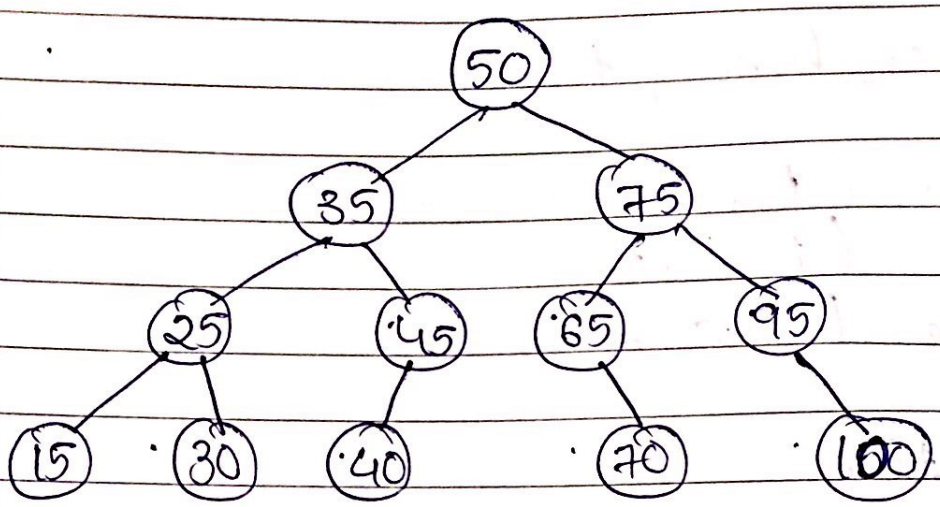
- H D I B J E K A L F M C N G O

- Preorder:

- A B D H I E J K C F L M G N O

- Postorder:

- H I D J K E B L M F N O G C A



→ Preorder:

~~50 25 15 30 3~~

- 50 35 25 15 30 45 40 75 65 70 95 100

→ Inorder:

- 15 25 30 35 40 45 50 65 70 75 95

→ Postorder:

- 15 30 25 40 45 35 70 65 100 95 75

- Traversal is possible only with recursion.

→ Inorder Algo^n:

Step-1 Ptr = Root
If (Ptr != NULL) then
Inorder (Ptr → LC)
visit (Ptr)
Inorder (Ptr → RC)
End if
Stop

~~Step-2~~

→ Preorder Algo^n:

Step-1 Ptr = Root
If (Ptr != NULL) then
visit (Ptr)
Preorder (Ptr → LC)
Preorder (Ptr → RC)
End if
Stop

→ Postorder Algo^n:

Step-1 Ptr = Root
If (Ptr != NULL) then
Postorder (Ptr → LC)
Postorder (Ptr → RC)
visit (Ptr)
Endif
Stop

Insertion in Binary Tree:

Input: key is the data contain of the key node after which a new node is to be inserted & item is the data element of the new node that has to be inserted.

→ Algorithm:

address of root

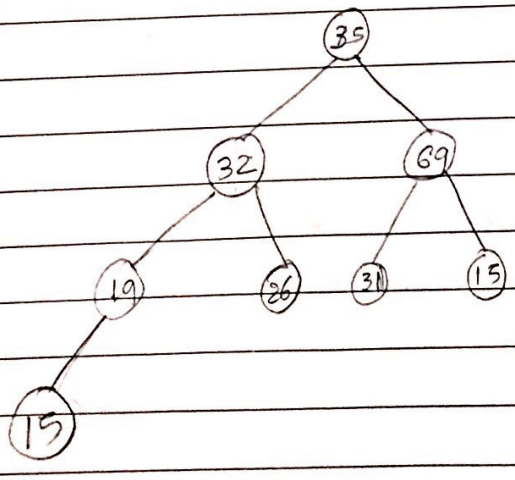
method / function of

```
Ptr = search_link ((Root, key)
If (Ptr = NULL) then
        Print "search is unsuccesfull"
        exit
end if
If (Ptr → LC = NULL) or (Ptr → RC = NULL)
    Read option (L) or (R)
    If (option = L) then
        If (Ptr → LC = NULL) then
            newnode = newnode (data)
            ptr → LC = newnode
        else
            print "Insertion is not possible as left ch
            exit
        endif
    else
        If (Ptr → RC = NULL) then
            newnode = newnode (data)
            ptr → RC = newnode
        else
            print "Insertion is not possible as right ch
            exit
        endif
```

```
else
    print "key node already has child"
endif
```

```
1. ptr = PTR0
   If (Ptr → data ≠ key) then
      If (Ptr → LC ≠ NULL)
         search_link (Ptr → LC)
      else
         return 0
      endif
      If (Ptr → RC ≠ NULL)
         search_link (Ptr → RC)
      else
         return 0
      end if
else
      return (Ptr)
end if
```

key = 19 ∴ item / data = 15

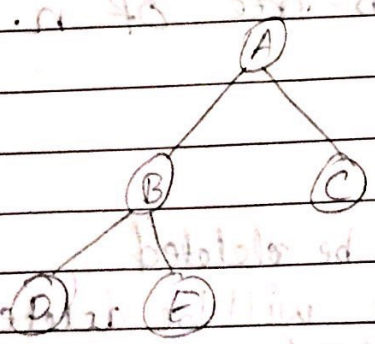


✱ Deletion in binary tree:

Step-1
```
Ptr = Root
If Ptr = NULL then
    print "tree is empty"
    exit
endif
Parent = SearchParent (Root, item)
If (Parent ≠ NULL) then
    Ptr1 = Parent → LC, Ptr2 = Parent → RC
    If (Ptr1 → data = item) then
        If (Ptr2 → LC = NULL) and (Ptr1 → RC = NULL) then
            parent → LC = NULL
        else
            Print "Node is not a leaf node"
        endif
    else
        If (Ptr2 → LC = NULL) and (Ptr2 → RC = NULL)
            parent → RC = NULL
        else
            Print "Node is not a leaf node"
        endif
    endif
else
    print "node with this data does not exit"
endif
Stop.
```

→ SearchParent:

```
Parent = PTR
If (Ptr → data ≠ item) then
    Ptr1 = Ptr → LC, Ptr2 = Ptr → RC
    If (Ptr1 ≠ NULL) then
        searchParent (Ptr1)
    else
        Parent = NULL
    endif
    If (ptr2 ≠ NULL) then
        searchParent (Ptr2)
    else
        Parent = NULL
    endif
else
    return Parent
endif
Stop
```



Types of binary tree:

(1) Expression tree
(2) heap tree
(3) BST (Binary search tree)
(4) AVL Tree ("height balance tree" actual name).
(5) Threaded binary tree
(6) Huffman tree
(7) Red black tree
(8) Splay tree
(9) decision tree

→ **Binary Search tree: (BST)**

- A binary tree T is term binary search tree if each node q of T satisfied the following property:

(1) The value at n is greater than every value in the left sub-tree of n & is less than every value in the right-sub-tree of n.

→ Deletion in BST:

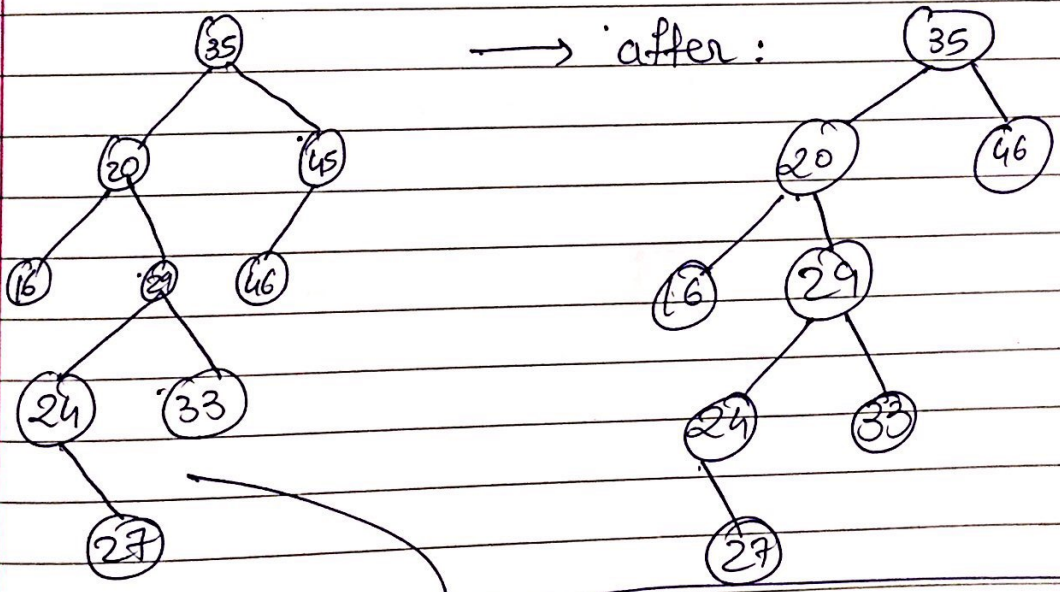
Case 1 - Leaf node will be deleted
Case 2 - Parent has One child node will be deleted
Case 3 - Parent has two child node will be deleted.

→ In one child node ~~the~~ ~~his~~ when the parent node will be deleted at that time address of child node will be at parent node place.

→ In two child node when the parent node will be deleted at that time right sub-tree in which the node which do not have left child will be replace and go to the place of parent node which is to be deleted.

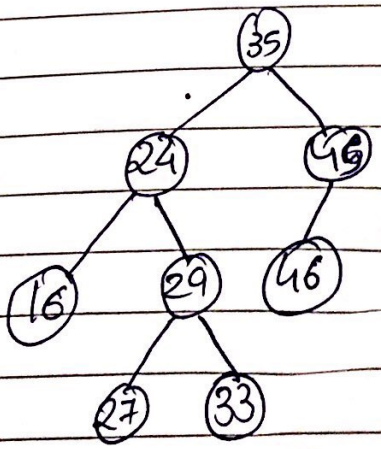
e.g:

→ one child:



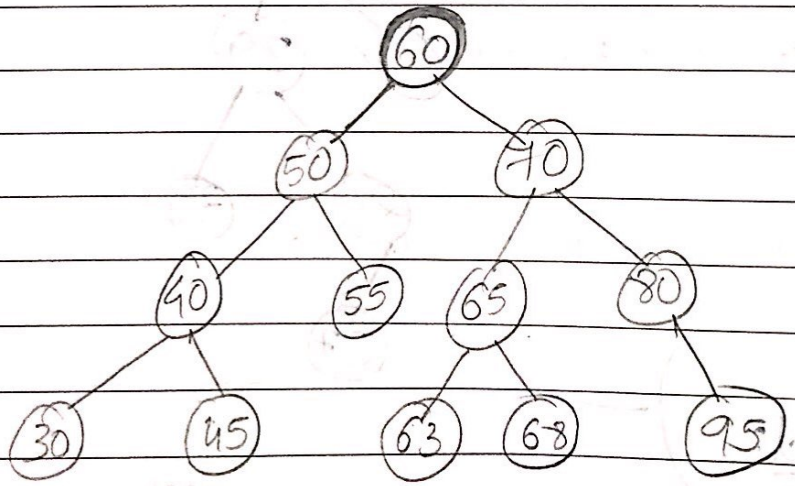
→ two child:

Same → after:



→ Searching in BST:

```
Ptr = Root, flag = false
while (Ptr ≠ NULL) and (flag = false) do
    case : item < Ptr → data
            Ptr = Ptr → Lchild
    case : Ptr → data = item
            flag = True
    case : item > Ptr → data
            Ptr = Ptr → Rchild
    end case
end while
if (flag = true) then
        print " item found at ", Ptr
else
        print " item does not exit"
endif
```

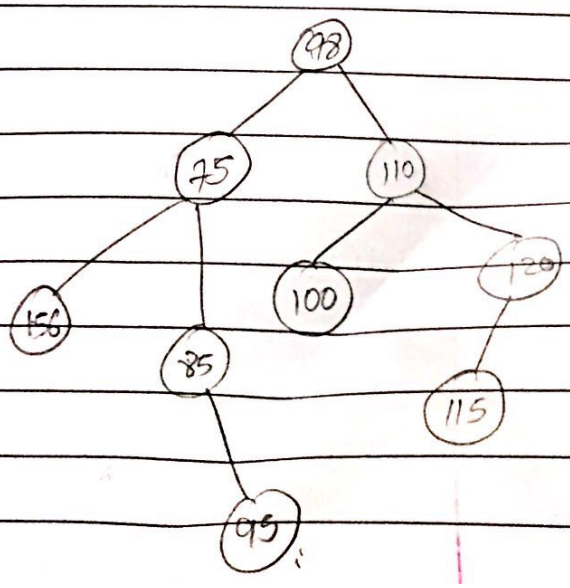


Insertion in BST:

```
Ptr = Root , flag = false
while (Ptr ≠ NULL) and (flag = false) do
    case: item < Ptr → data
        Ptr1 = Ptr
        Ptr = Ptr → Lchild
    case: item > ptr → data
        Ptr1 = Ptr
        Ptr = Ptr → Rchild
    case: item = Ptr → data
        flag = true
        Print "item already exist"
        exit
end case
end while
If Ptr = NULL then
    newnode = new node (item);
    If (Ptr1 → data < item) then
        Ptr1 → Rchild = newnode
    else
        Ptr1 → Lchild = newnode
    end if
end if
```

Ptr1	Ptr
A(98)	A(98)
A(110)	A(110)
A(120)	A(120)
	NULL

Ptr1	Ptr
A(98)	~~A(98)~~
A(75)	A(75)
A(85)	~~A(85)~~
	NULL



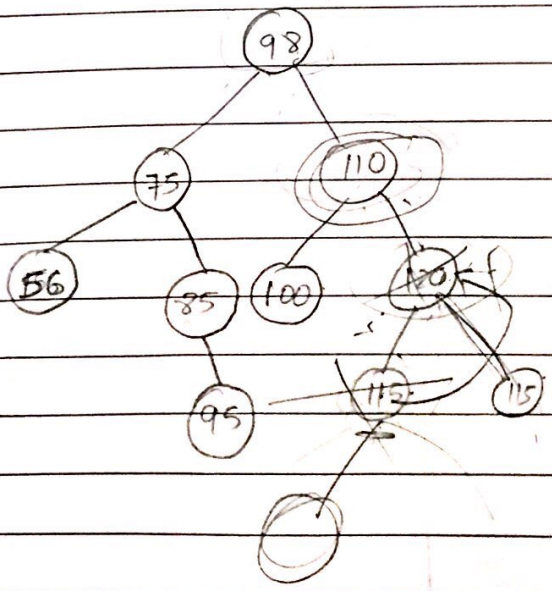
→ Deletion of BST:

→ delete_BST ()

```
{ Ptr = Root , flag = False
while ( ptr ≠ NULL ) and (flag = false) do
    case: item < ptr → data
            parent = ptr
            ptr = ptr → Lchild
    case: item > ptr → data
            parent = ptr
            ptr = ptr → Rchild
    case: ptr → data = item
            flag = True
    endcase
end while
If (flag = false) then
    print " item does not exist: No deletion"
    exit
end if
If ( ptr → Lchild = NULL) and (ptr → Rchild = NULL) then
        case = 1
else
    If ( ptr → Lchild ≠ NULL) and ( ptr → Rchild ≠ NULL) then
        case = 3
    else
        case = 2
    endif
endif
If (case = 1) then
    If ( parent → Lchild = Ptr) then
        parent → Lchild = NULL
    else
        parent → Rchild = NULL
```

Ptr	parent
~~95~~	95
~~110~~	110
~~120~~	120
115	

```
        endif
        returnNode(ptr)
endif
If (case=2) then
    If (parent → Lchild = ptr) then
        If (ptr → Lchild = NULL) then
            parent → Lchild = ptr → Rchild
        else
            parent → Lchild = ptr → Lchild
        endif
    else
        If (parent → Rchild = ptr) then
            If (ptr → Lchild = NULL) then
                parent → Rchild = ptr → Rchild
            else
                parent → Rchild = ptr → Lchild
            endif
        endif
        returnNode(ptr)
endif
If (case = 3)
    ptr1 = SUCC(ptr)
    item1 = ptr1 → data
    Delete_BST (item1)
    ptr → data = item1
endif
Stop
```



Ptr	Parent
~~98~~	98
~~110~~	110
120	

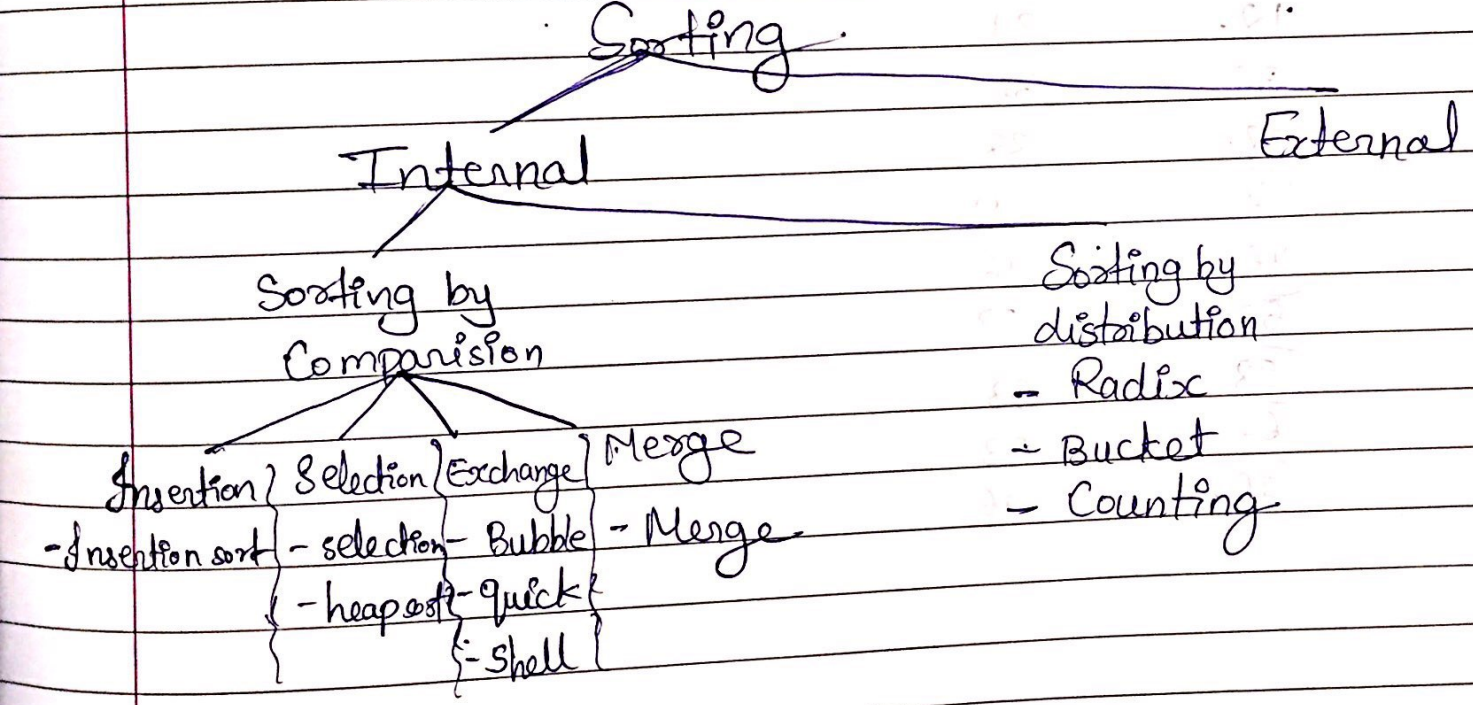
→ Algo^n of SUCC:

```
ptr1 = PTR → Rchild
If (ptr1 ≠ NULL) then
    while (ptr1 → Lchild ≠ NULL) do
        ptr1 = ptr1 → Lchild
    end while
endif
return (ptr1)
Stop.
```

Sorting : Terms

Internal Sort :

- Internal Sort
- External Sort
- Order Sort
  - Ascending
  - Descending
  - Lexicographic → abc, abb (abb will come first)
  - Collating Sequence
- Swap
- Stable Sort
- Inplace Sort



→ Bubble sort:

1	2	3	4	5	6	7
44	36	21	21	21	21	21
36	21	36	36	36	36	36
21	44	44	44	44	38	18
58	58	52	52	46	18	38
68	52	58	46 52	38 46	44	44
52	68	46 58	38 52	18	46	46
76	46	38 52	18 52	52	52	52
46	38	18	58	58	58	58
38	18	68	68	68	68	68
18	76	76	76	76	76	76

8	9
21	18
18	21
36	36
38	38
44	44
46	46
52	52
58	58
68	68
76	76

→ Selection Sort:

	1	2	3	4	5	6	7
~~70~~	12	12	12	12	12	12	12
36	36	20	20	20	20	20	20
44	44	44	36	36	36	36	36
12	70	70	70	44	44	44	44
89	89	89	89	89	52	52	52
20	20	36	44	70	70	58	58
58	58	58	58	58	58	70	64
64	64	64	64	64	64	64	70
76	76	76	76	76	76	76	76
52	52	52	52	52	89	89	89

min= ~~70~~, 36, 12

min= ~~36~~, 20

min= ~~44~~, 36

min= ~~70~~, 44

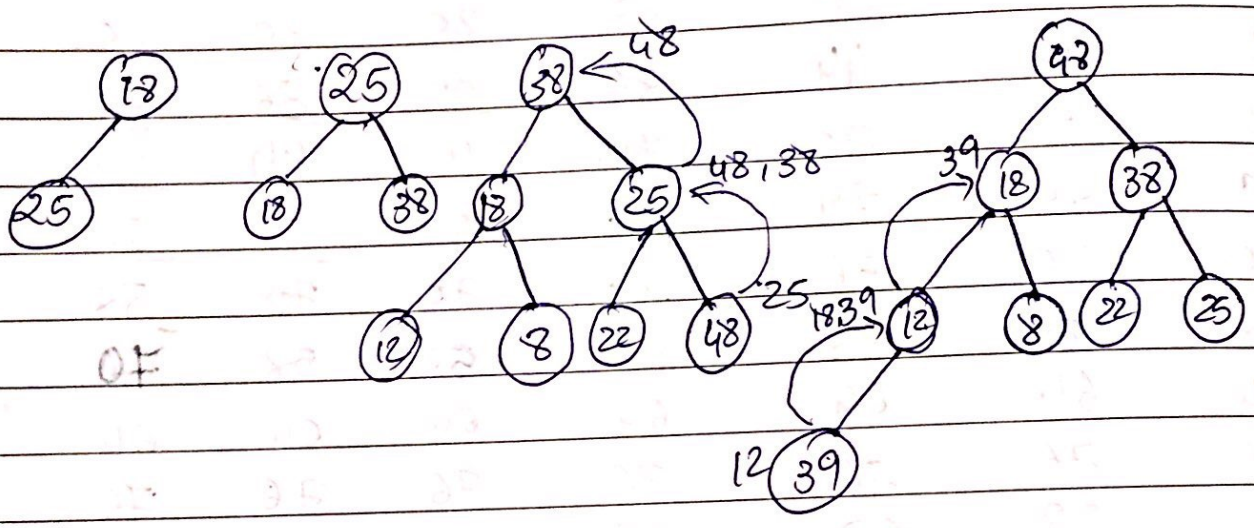
min= ~~89~~, ~~70~~, ~~58~~, 52

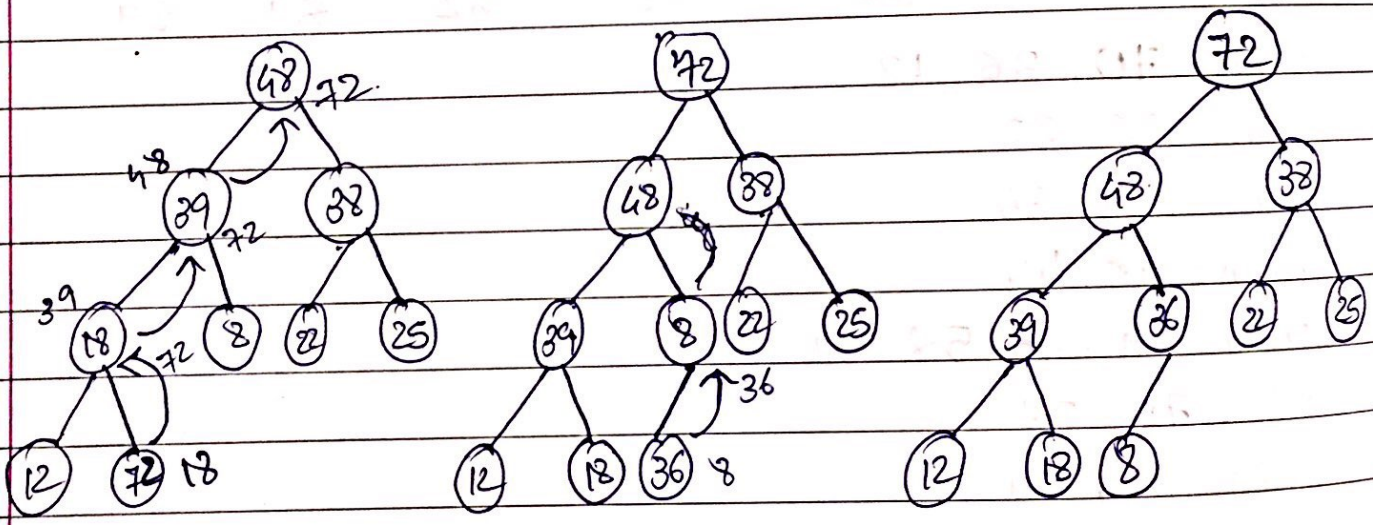
min= ~~70~~, 58

min= ~~70~~, 64.

→ Heap · Sort:

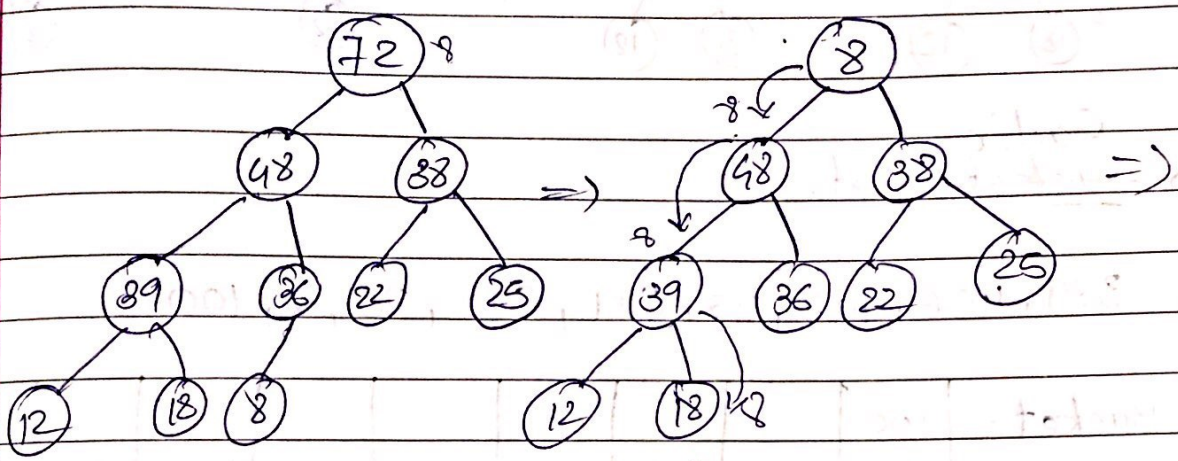
18, 25, 38, 12, 8, 22, 48, 39, 72, 36

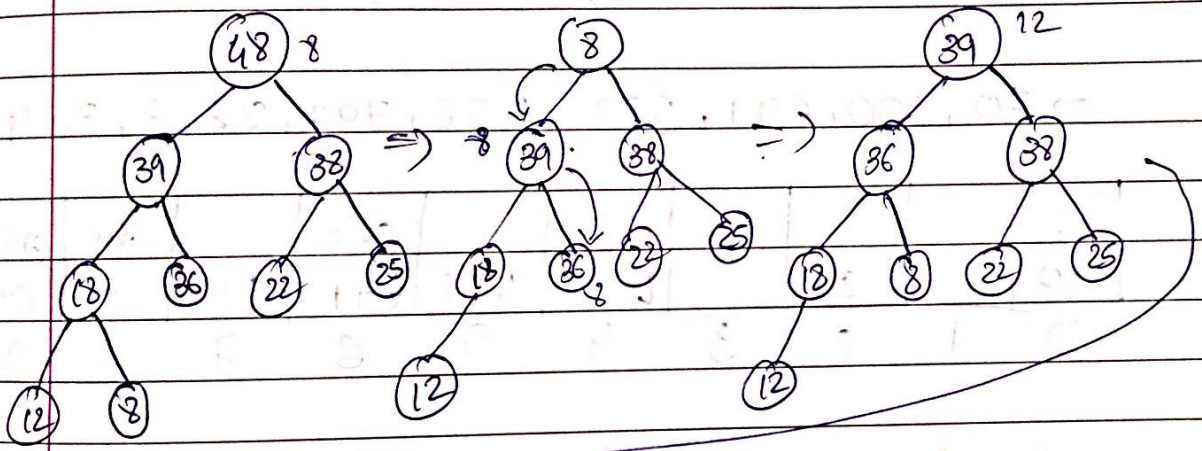


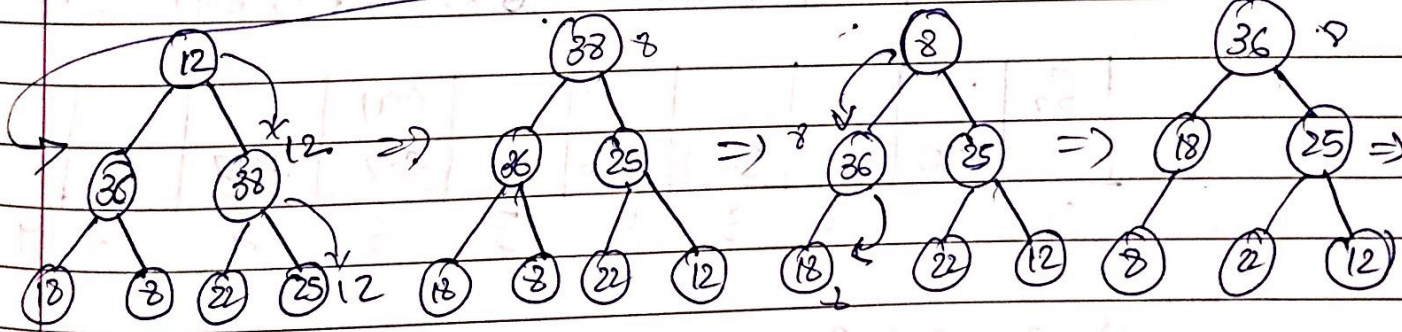


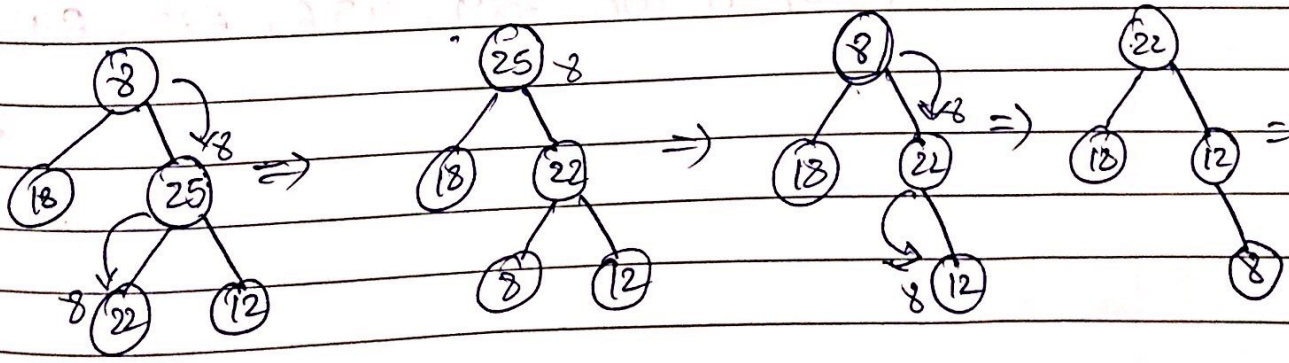
replace = 72, 48, 39, 38, 36, 25, 22, 18, 12, 8

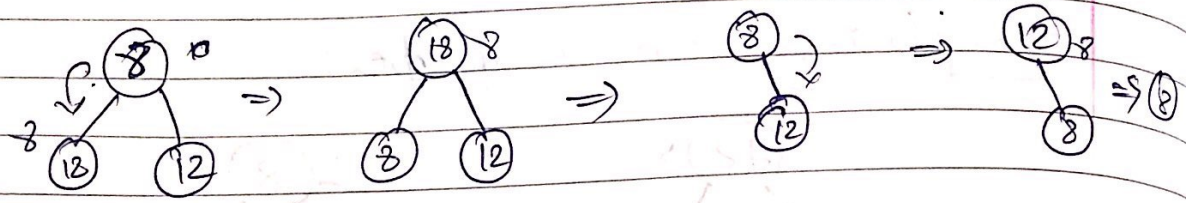
→ Replacing from heap tree:











→ Radix Bucket Sort:

389, 456, 797, 28, 691, 682, 70, 8, 100

Bucket –

0	1	2	3	4	5	6	7	8	9
100 70	691	682				456	797	8 28	389

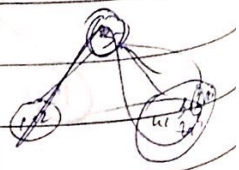
→ 70, 100, 691, 682, 456, 797, 28, 8, 389.

0	1	2	3	4	5	6	7	8	9
8 100		28		1	456	~~682~~ ~~691~~	70	389 682	797 691

→ 100, 8, 28, 456, 70, 682, 389, 691, 797

0	1	2	3	4	5	6	7	8	9
70 28 8	100		389	456		691 682	797		

→ 8, 28, 70, 100, 389, 456, 682, 691, 797



⇒ Insertion Sort:

1	2	3	4	5	6	7
70	44	12	12	12	12	12
44	70	44	36	36	36	36
12	12	70	44	44	44	44
36	36	36	70	69	69	69
69	69	69	69	70	70	52
86	86	86	86	86	52	70
52	52	52	52	52	86	86
77	77	77	77	77	77	77
18	18	18	18	18	18	18

8	9	10	11
12	12	12	12
36	36	36	18
44	44	44	36
52	52	52	44
69	69	69	52
70	70	70	69
86	77	77	70
77	86	18	77
18	18	86	86

→ h-increament → h value will be always in odd terms.

Shell Sort → 21, 72, 43, 94, 85, 16, 67, 38, 59, 91, 32, 73, 24, 45, 56, 60.

h=7

A[1], A[8], A[15] = {21, 38, 56}
A[2], A[9], A[16] = {59, 60, 72}
A[3], A[10] = {43, 91}
A[4], A[11] = {32, 94}
A[5], A[12] = {73, 85}
A[6], A[13] = {16, 24}
A[7], A[14] = {45, 67}

⇒ 21, 59, 43, 32, 73, 16, 45, 38, 60, 91, 94, 85, 24, 67, 56, 72

h=5

A[1], A[6], A[11], A[16] = {21, 16, 94, 72} = {16, 21, 72, 94}
A[2], A[7], A[12] = {59, 45, 85} = {45, 59, 85}
A[3], A[8], A[13] = {43, 38, 24} = {24, 38, 43}
A[4], A[9], A[14] = {32, 60, 67} = {32, 60, 67}
A[5], A[10], A[15] = {73, 91, 56} = {56, 73, 91}

~~⇒ 21, 16, 94, 72, 59,~~
~~⇒ 21, 59, 43, 32, 73, 16, 45, 38, 60, 91, 94, 85, 24, 67, 56, 72~~

⇒ 16, 45, 24, 32, 56, 21, 59, 38, 60, 73, 72, 85, 43, 67, 91

h=3

A[1], A[4], A[7], A[10], A[13], A[16] = {16, 32, 59, 73, 43, 94}
A[2], A[5], A[8], A[11], A[14] = {95, 56, 38, 72, 67}
A[3], A[6], A[9], A[12], A[15] = {24, 21, 60, 85, 91}

= {16, 32, 43, 59, 73, 94}
= {38, 45, 56, 67, 72}
= {21, 24, 60, 85, 91}

⇒ 16, 38, 21, 32, 45, 24, 43, 56, 60, 59, 67, 85, 73, 72, 67, 91

~~h=1~~

A[1], A[4], A[7]

⇒ 16, 38, 21, 32, 45, 24, 43, 56, 60, 59, 67, 85, 73, 72, 91, 94.

h=1

A[1], A[2], A[3], A[4], A[5], A[6], A[7] ---- A[16] = {16, 38, 21, 32, 45, 24, 43, 56, 60, 59, 67, 85, 73, 72, 91, 94} ⇒

= {16, 21, 24, 32, 38, 43, 45, 56, 59, 60, 67, 72, 73, 85, 91, 94}

→ Counting Sort: (less than equal to

4	6	3	5	9	2

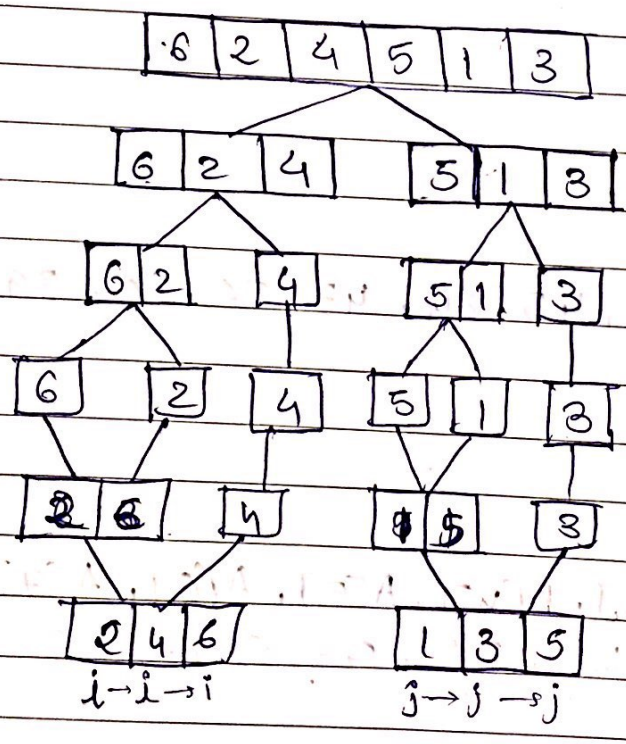
Count

3	5	2	4	6	1

Resultant

2	3	4	5	6	9

→ Merge Sort:



1	2	3	4	5	6

→ Quick Sort:

55 88 22 99 44 11 66 77 33

L R

33 22 99 44 55 11 66 77 88

L L L R R R R R

{33 11 22} 44 {55} {99 66 77 88}

L L R R L L L R R

{22, 11} 33 44 55 {88 66 77} 99

11 22 33 44 55 66 77 88 99.

New Unit

→ ~~Search~~ linear Search & Non-linear Search:

→ Ordered list Algo^{n}:

```
i=l, found=0, loc=-1
while (key>=a[i]) and (found=0) do
    compare (a[i], key) = True then
            loc=i
            found=1
    else
        i=i+1
    end if
end while
Stop if (found=0) then
        print "Not found"
    else
        print "found"
    end if
Stop
```

→ Binary Search:

```
L=1, U=n, flag=false
while (flag ≠ True) & (L<U) do
    mid = (L+U)/2
    If K=a[mid] then
        print "Successful"
        flag = True
        Return (mid)
    Endif
    If (K<A[mid]) then
        U = mid-1
    else
        L = mid+1
    end if
end while
If (flag = false) then
    print "unsuccessful"
    Return -1
end if
```

1	2	3	4	5	6	7	8	9	10	11
5	15	25	35	40	45	50	55	65	90	2

L

key = ~~65~~ 15

L	U	mid
1	12	6
7		9
1	12	6
1	5	3
1	2	1
2	2	2

} 65