DATA STRUCTURES

Data – It is a simple collection of facts, figures, numbers, characters, symbols, etc whatever we need to store in memory.
Eg: 25.

Abstract Datatype :

→ It is a most suitable datatype chosen to store the given data in well-organized manner.
Eg: To store the data '45', we can choose 'int' datatype.

Data Structures :

→ It is a memory created using abstract datatype to store the given data is called data structures.

Data Abstraction :

→ Extracting the useful information from a database, and visualising in local device in a well-defined manner called data abstraction.

Data Model :

→ A group of related data stored in a tabular manner is called data model.

Eg:

Ac no.	Name	Balance
100	Alok	5000
101	Sam	50000
102	Santosh	500000
103	Mohit	25000

→ To store the above model, the most suitable datatype or abstract datatype is –

```
struct bank
{
   int ac no;
   char name[10];
   float balance;
}
```

→ Using 'struct bank', we can allocate memory as struct bank A[5];

	Ac	Name	Balance
0			
1			
2			
3			
4			

→ Now the memory A[5] which can store the data model is called data structure.

TYPES OF DATA STRUCTURE

They are 2 types:

1) Primitive type DS
2) Non-Primitive type DS

Primitive type DS

→ Using primary datatypes, when we allocate a memory, then it is called primitive type DS.

Non-Primitive type DS

→ Using derived and user-defined datatypes, when we allocate memory then it is called non-primitive type DS.

→ It is again of 2 types:

1) Linear DS
2) Non-linear DS

A. Linear Data Structure

→ The data structure which occupies memory in serial manner and allows operations linearly is called linear data structure.

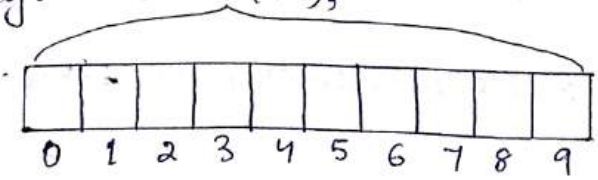
→ They are:

1) Array
2) Linked list
3) Stack
4) Queue

1. Array

→ A group of consecutive memory locations having same datatype is called an array.

Eg: int A[10];

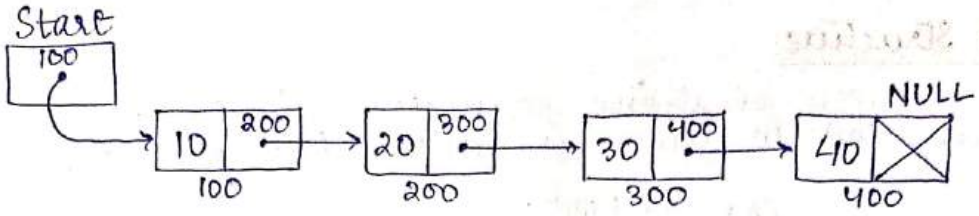


2. Linked List

→ A group of non-consecutive memory locations linked in serial manner and allows operations serially is called linked list.

→ In linked list, each memory location is called a NODE.

Standard Diagram :



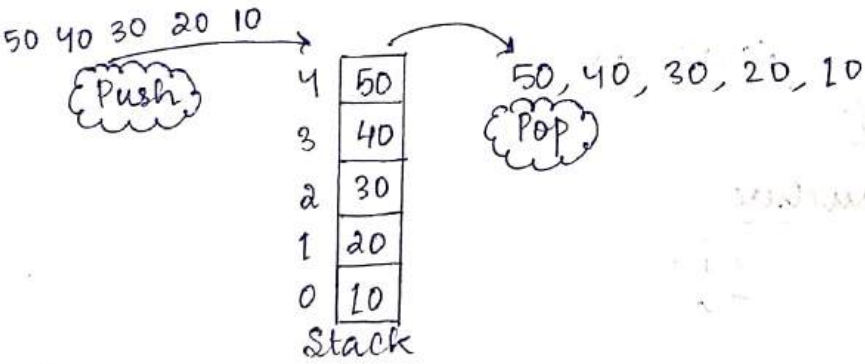
3. Stack

→ A linear data structure which implements last in first out concept (LIFO) is called stack. A stack can be implemented using array and linked list.

Eg: a) Stack of books,

b) Stack using array

Eg: int a[5];



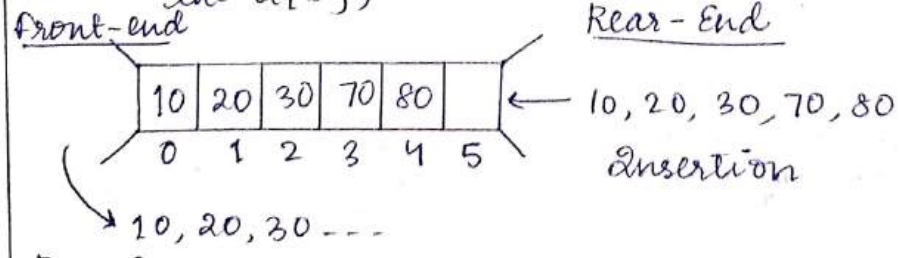
4. Queue

→ A linear DS which implements first in first out concept is called queue.

→ The queue concept can be implemented using array and linked list.

Eg: Queue using array

int a[6];



B. Non-linear DS

→ A group of memory locations connected not in serial manner is called non-linear D.S.

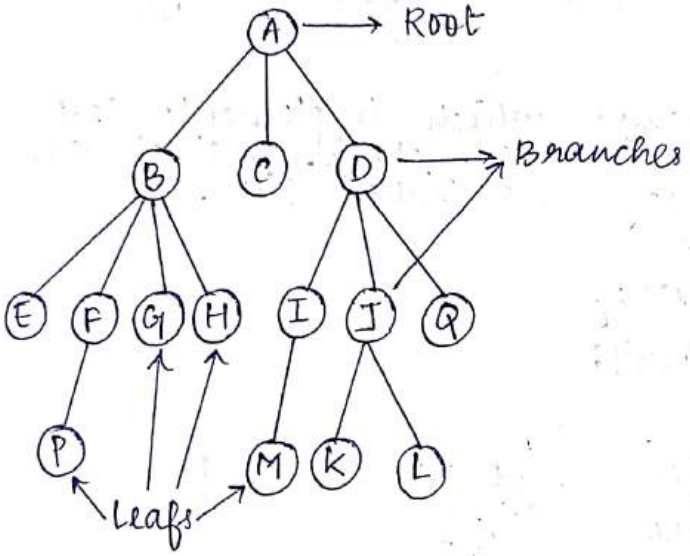
→ They are :

1) Tree
2) Graph

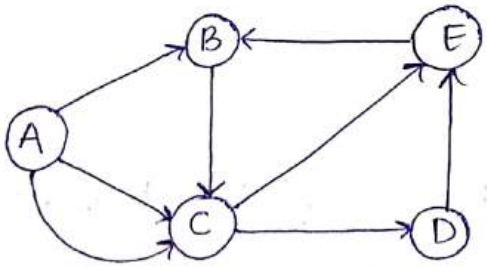
1. Tree Data Structure

→ A group of memory locations connected in hierarchial manner from top to down is called tree data structure.

Eg:



2. Graph Data Structure



→ A collection of memory locations in which if there is random relationships exist then it is called graph.

→ To implement the memory representation of data structure and to understand different operations on them, algorithm can be used.

Algorithm

→ Representing the problem statements using simple english in countable number of steps without confusion is called algorithm.

Standard Notations

→ To maintain uniformity, we use common notations as follows :-

1. Start at the beginning
   Stop at the end
2. Documentation must be within [and]
3. Declaration can be using - let
   Eg: let x, y
4. Assignment can be using = or := or ←
   Eg: x = 25 OR x := 25 OR x ← 25
5. To input we can use Input / Accept / Read
6. To output we can use Display / Write / print
7. All the mathematical symbols can be used.
8. Step nos to be followed for each statement.

→ These notations allows to write algorithm similar to pseudocode.

→ The algorithm statements are of 3 types:
   a) Sequential statements
   b) Selection control statements
   c) Loop control statements

A. Sequential Statements

Q. Write an algorithm to input radius of a circle. Find the area and perimeter of it.

Step 1: Start
Step 2: Let rad, area, peri
Step 3: Display "Enter Radius"
Step 4: Input rad
Step 5: Area = 3.14 * rad * rad
Step 6: Peri = 3.14 * 2 * rad
Step 7: Display "Area=", area
Step 8: Display "Perimeter =", peri
Step 9: Stop

O/P Screen

Enter Radius = 4.5
Area = 63.58
Perimeter = 28.26

Q. Write an algorithm to find simple interest.

Step 1: Start
Step 2: Let principle, rate, time, SI
Step 3: Display "Enter principle, time, rate, SI".
Step 4: Input principle, rate, time
Step 5: SI = (Principle * rate * time) / 100
Step 6: Display "Simple interest =", SI
Step 7: Stop

B. Selection Control Statements

→ They are :

1) simple if
2) if -- else
3) else if ladder
4) nested if

1) Simple if

Syntax :
```
if (condition) then
    statements
[end of if]
```

2) If -- else

Syntax :
```
if (condition) then
    statement block 1
else
    statement block 2
[end of if]
```

Eg: Write an algorithm to find greatest among two unequal numbers.

```
Step 1: Start
Step 2: Let x, y
Step 3: Display "Enter 2 unequal nos".
Step 4: Input x, y
Step 5: if (x > y) then
  Step 5.1: Display "Greatest is ", x
Step 6: else
  Step 6.1: Display "Greatest is ", y
  [end of if]
Step 7: Stop
```

3) Else if ladder

Syntax :
```
if (condition1) then
    statement block 1
else if (condition 2) then
    statement block 2
else
    statement block K
[end of if]
```

Eg: Write an algorithm to find the greatest among three unequal numbers using else if ladder.

[Finding greatest among 3 nos]

Step1: Start
Step2: Let x, y, z
Step3: Display "Enter 3 unequal nos"
Step4: Input x, y, z
Step 5: if (x > y AND x > z) then
  Step 5.1: Display "Greatest is", x
Step 6: else if (y > x AND y > z) then
  Step 6.1: Display "Greatest is", y
Step 7: else
  Step 7.1: Display "Greatest is", z
Step 8: [end of if]
Stop

4) Nested if

Syntax:

```
if (condition1) then
    if (condition 2) then
        statement, block 1
    else
        statement, block 2
    [end of if]
[end of if]
```

Eg: Write an algorithm to find the greatest among three unequal numbers using nested if.

[Find greatest among 3 nos]

Step 1: Start
Step 2: Let x, y, z
Step 3: Display "Enter 3 unequal nos"
Step 4: Input x, y, z
Step 5: if (x > y) then
  Step 5.1: if (x > z) then
    Step 5.1.1: Display "Greatest is", x
  Step 5.2: else
    Step 5.2.1: Display "Greatest is", z
  [end of 5.1]
Step 6: else
  Step 6.1: if (y > z) then
    Step 6.1.1: Display "Greatest is", y
  Step 6.2: else
    Step 6.2.1: Display "Greatest is", z

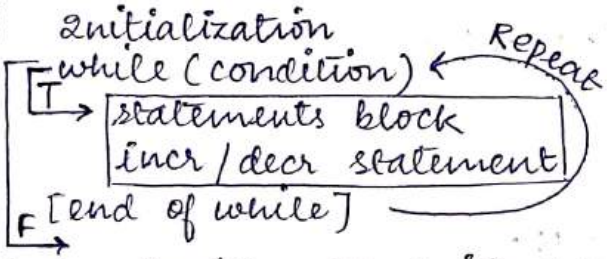
[end of if – 6.1]
[end of if – 5]
Step 7: stop

C. Loop Control Statements

→ The looping statements available in algorithm are:
1) while
2) for
3) repeat until

1. While:

Syntax: Initialization



Eg: Write an algorithm to print 1 to 100 using while loop.
[Print 1 to 100]
Step 1: Start
Step 2: Let i
Step 3: i = 1
Step 4: while (i <= 100)
Step 4.1: Display i
Step 4.2: i = i + 1
[end of while]
Step 5: Stop

2. For:

Syntax: for (initial value to final value), incr / decr
statements
[end of for]

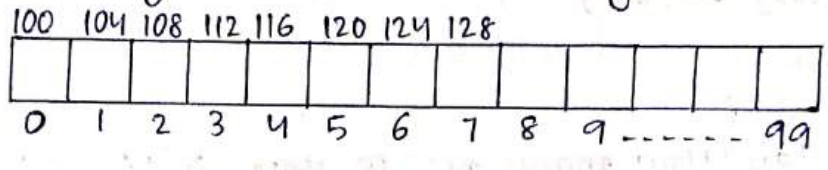
Eg: Write an algorithm to print 1 to 100 using for loop.
[Print 1 to 100]
Step 1: Start
Step 2: Let i
Step 3: for (i = 1 to 100), incr by 1
Step 3.1: Display i
[end of for]
Step 4: stop

ARRAY

→ Array is of 2 types:
1) 1-D Array
2) Multi-dimensional array

1-D Array

Consider an array A[100] to store integers



Base = 100
Size of each memory $w = 4$ bytes
Then to find address of ith location = $\boxed{Base + w * i}$
Eg: $i = 7$
Then address of $a[7] = 100 + 4 * 7$
$= 128$

Q. Find the address of $i = 10$, when base = 100 and size of memory $w = 4$ bytes.
A- Address of $A[10] = 100 + 4 * 10$
$= 140$

Q. Suppose an array Q[20] of double datatype. Assume base address = 1000, size of memory $w = 8$ bytes. Find address of Q[12].
A- $Q[12] = 1000 + 8 * 12$
$= 1000 + 96$
$= 1096$

→ The common operations on array are:
1) Creation
2) Traversion
3) Insertion
4) Deletion
5) Searching
6) Sorting
7) Merging
8) Concatenating

Consider an array a[100] for performing different operations on it.

Assume lb represents lower bound and ub represents upper bound. (lb → index of first element)
ub → index of last element

A. Creation Algorithm

Create (a[100], lb, ub)

Step 1 : Start
Step 2 : Let n, i
Step 3 : Display "How many nos. to store initially"
Step 4 : Input n
Step 5 : i = 0
Step 6 : while (i < n)
  Step 6.1 : Input a[i]
  Step 6.2 : i = i + 1
  [end of while]
Step 7 : lb = 0, ub = n - 1
Step 8 : Stop

B. Traversion

→ Visiting all the elements of an array, that is from lb to ub serially is called traversion.

Traversion (a[100], lb, ub)

Step 1 : Start
Step 2 : Let i
Step 3 : Display "All the elements are"
Step 4 : For (i = lb to ub), incr by 1
  Step 4.1 : Display a[i]
  [end of for]
Step 5 : Stop

Using while loop
Step 4 : i = lb
Step 5 : while (i <= ub)
  Step 5.1 : Display a[i]
  Step 5.2 : i = i + 1
  [end of while]

C. Searching

Search (a[100], lb, ub)

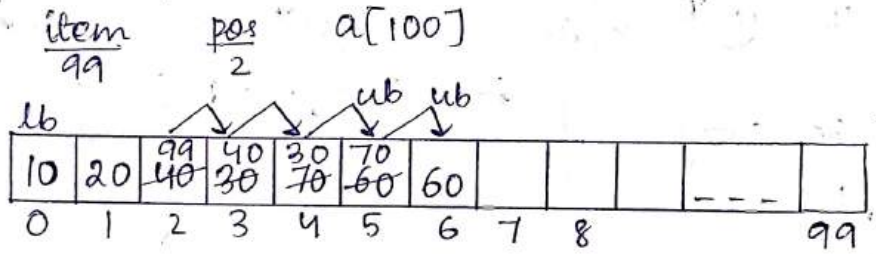
Step 1 : Start
Step 2 : Let i, item
Step 3 : Display "Enter the item to search"
Step 4 : Input item
Step 5 : For (i = lb to ub), incr by 1
  Step 5.1 : if (a[i] = item), then

Step 5.1.1 : Display "Item found at index", i
[end of if]
[end of for]
Step 6 : Stop

D. Insertion

Given an item and position for insertion.
Insertion (a[100], lb, ub, item, pos)

Step 1 : Start
Step 2 : Let i
Step 3 : For (i = ub to pos), decr by 1.
Step 3.1 : a[i+1] = a[i]
[end of for]
Step 4 : a[pos] = item
Step 5 : ub = ub+1
Step 6 : Stop

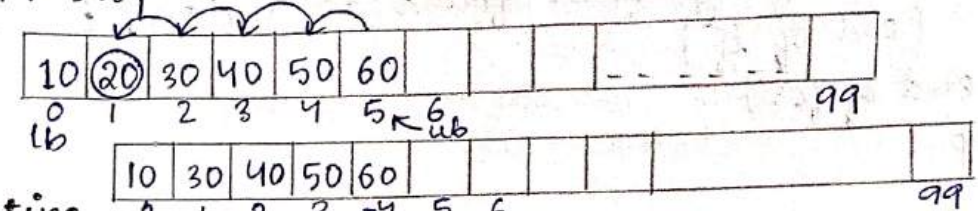


E. Deletion

Given an item to search and delete
Deletion (a[100], lb, ub, item)

Step 1 : Start
Step 2 : Let i
Step 3 : For (i = lb to ub), incr by 1
Step 3.1 : if (a[i] = item), then
Step 3.1.1 : Break
[end of if]
[end of for]
Step 4 : if (i > ub), then
Step 4.1 : Display "Item not found"
Step 5 : else
Step 5.1 : while (i <= ub)
Step 5.1.1 : a[i] = a[i+1]
Step 5.1.2 : i = i+1
[end of while]
[end of if]
Step 6 : ub = ub-1

Step 7: Stop



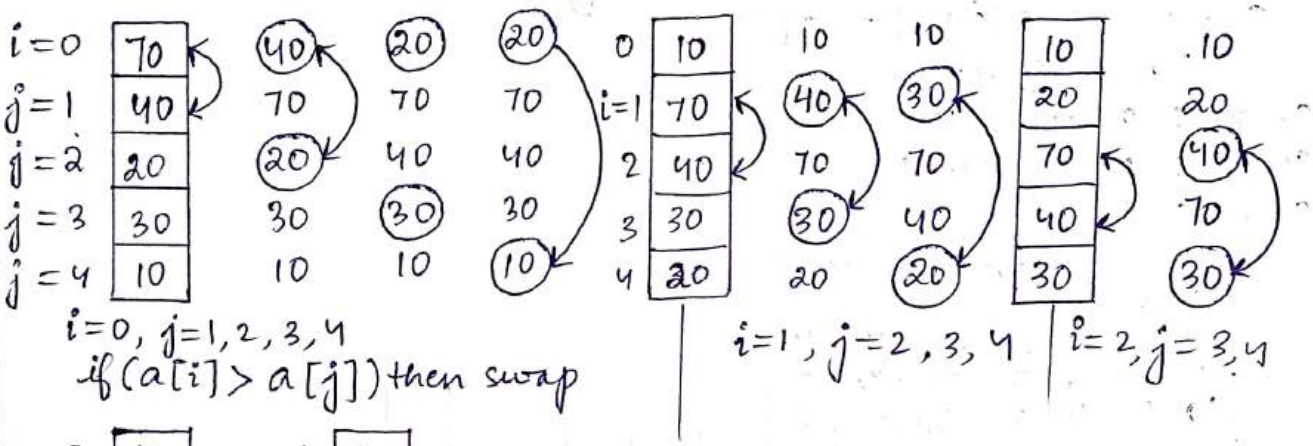
F. Sorting

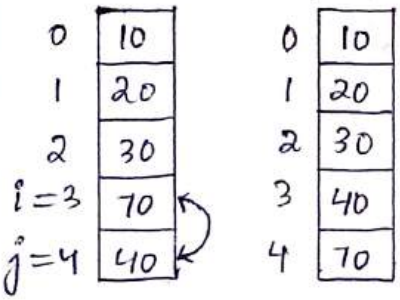
→ Arranging the elements in ascending or descending order is called sorting.

* Selection Sort:

Here each element is compared with the neighbour element for swapping.

Consider an array a[5]





i=3, j=4

Sorting (a[100], lb, ub)

Step 1: Start

Step 2: Let i, j, temp

Step 3: for (i = lb to ub-1), incr by 1.

Step 3.1: for (j = i+1 to ub), incr by 1.

Step 3.1.1: if (a[i] > a[j]) then

Step 3.1.1.1: temp = a[i]

3.1.1.2: a[i] = a[j]

3.1.1.3: a[j] = temp

[end of if]

[end of for - 3.1]

[end of for - 3]

step 4: Stop

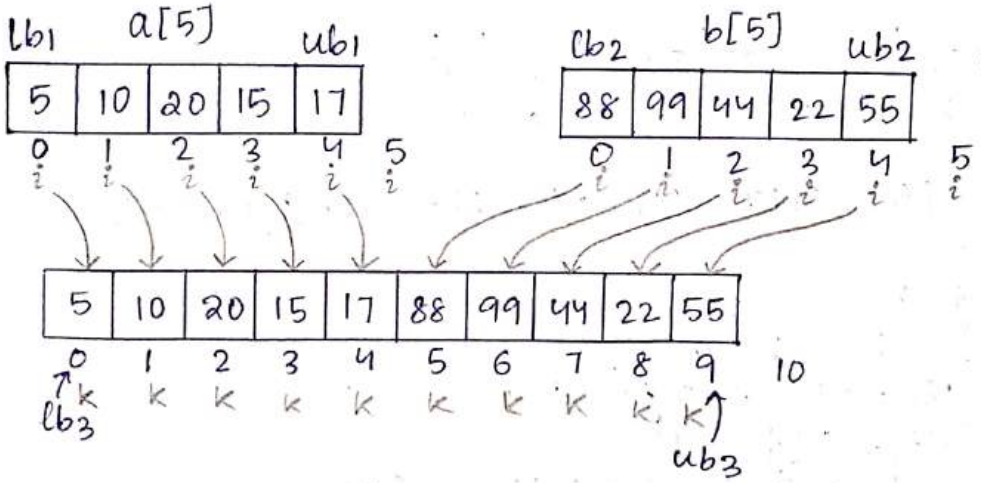
G. Concatenation

→ Here, we need to consider 2 arrays along with elements are given for joining together.

Given two arrays a[5] and b[5] having elements and we need to store together into 3rd array c[10].

Concatenate (a[5], b[5], c[10], lb1, ub1, lb2, ub2)

Step 1 : Start
Step 2 : let i, k, lb3, ub3
Step 3 : i = lb1, k = 0
Step 4 : while (i <= ub1)
  Step 4.1 : c[k] = a[i]
  Step 4.2 : i = i+1, k = k+1
    [end of while]
Step 5 : i = lb2
Step 6 : while (i <= ub2)
  Step 6.1 : c[k] = b[i]
  Step 6.2 : i = i+1, k = k+1
    [end of while]
Step 7 : lb3 = 0, ub3 = k-1
Step 8 : Stop



H. Merging

→ Joining 2 array elements into a resultant array in sorted manner is called merging.

→ Merging is possible in two situations :
i) when two unsorted list of elements given
ii) when two sorted list of elements given

When two unsorted list of elements given:

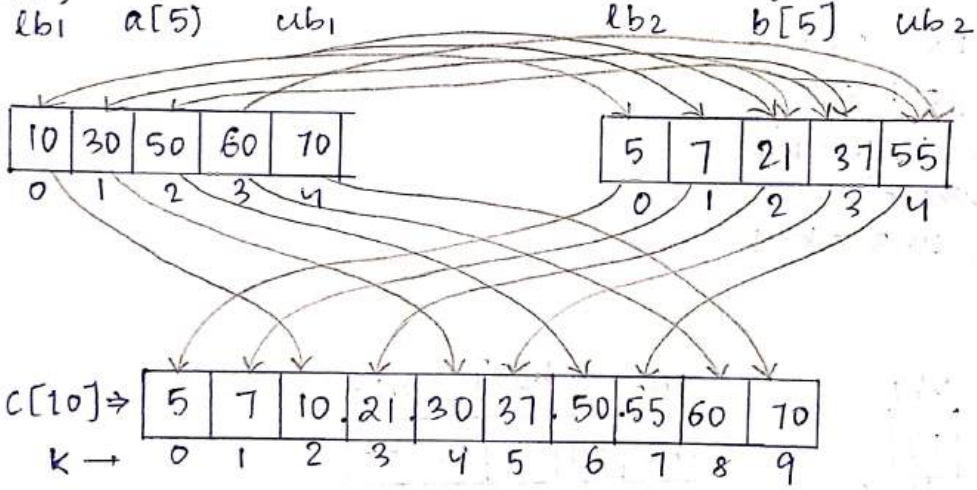
→ Here, we need to concatenate and apply sorting.

Merging (a[5], b[5], c[10], lb1, ub1, lb2, ub2, lb3, ub3)

Step 1: Start
Step 2: concatenate (a[5], b[5], c[10], lb1, ub1, lb2, ub2, lb3, ub3)
Step 3: sorting (c[10], lb3, ub3)
Step 4: Stop

When two sorted list of elements given:

Suppose given two array a[5], b[5] and c[10].
a[5] having elements from lb1 to ub1.
b[5] having elements from lb2 to ub2.
c[10] shall contain resultant list of elements.



Merging (a[5], b[5], c[10], lb1, ub1, lb2, ub2, lb3, ub3)

Step 1 : Start
Step 2 : let i, j, k
Step 3 : i = lb1, j = lb2, k = 0
Step 4 : while (i <= ub1 AND j <= ub2)
  Step 4.1 : if (a[i] < b[j]), then
    Step 4.1.1 : c[k] = a[i], k = k+1, i = i+1
  Step 4.2 : else
    Step 4.2.1 : c[k] = b[j], k = k+1, j = j+1
    [end of if]
  [end of while]
Step 5 : while (i <= ub1)
  Step 5.1 : c[k] = a[i], k = k+1, i = i+1
  [end of while]
Step 6 : while (j <= ub2)
  Step 6.1 : c[k] = b[j], k = k+1, j = j+1
  [end of while]

Step 7 : $lb_3 = 0$, $ub_3 = k-1$
Step 8 : Stop

Two-dimensional Array

→ A 2-D array occupies the memory just similar to 1-D array. The memory representation of a 2-D array is just like 1-D array, that is, in 2 ways :-

1) Row major order
2) Column major order

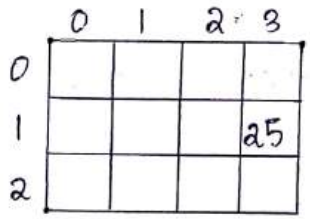
Row major order:

→ when the memory is occupied row by row serially, then it is called row major order.

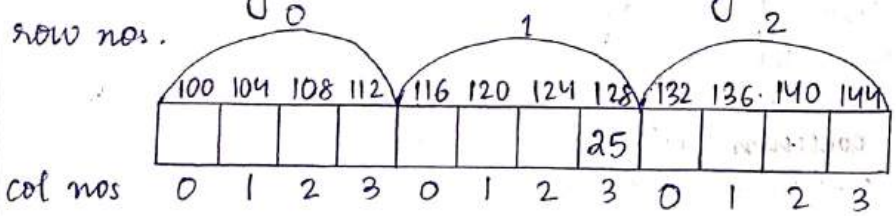
Column major order :

→ when the memory is occupied column by column serially, then it is called column major order.

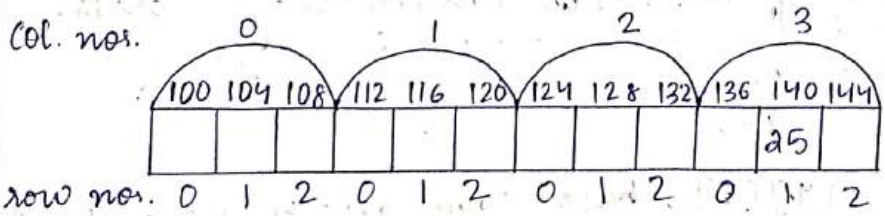
Consider a matrix a[3][4] which contains integers with size of each memory = 4 bytes and base address is 100.



In row major order the memory will be as :



In column major order the memory will be as :



Here, we can observe the address of a[1][3] which contains 25 is different in row major order and column major order, i.e, in row major order address of a[1][3] is 128 and in column major order it is 140.

→ In row major order, to find the address of any location, the formula given is address of $a[i][j] = Base + w*(n*i+j)$

where Base is Base address
w is size of each memory
m is row size
n is column size
i is row index
j is column index

→ In above example : Given a matrix a[3][4], base address = 100, size of each memory is 4 bytes.

Q. In row major order, find address of a[1][3].

Sol- Base = 100, w = 4 bytes, m = 3, n = 4, i = 1, j = 3

Address of a[1][3] = Base + w * (n * i + j)
= 100 + 4 * (4 * 1 + 3)
= 128

Q. Given a matrix a[4][5], size of each memory = 8 bytes, base address is 1000. Find address of a[3][1] in row major order.

Sol- Base = 1000, w = 8 bytes, m = 4, n = 5, i = 3, j = 1.

Address of a[3][1] = 1000 + 8 * (5 * 3 + 1)
= 1128

→ In column major order, to find the address of any location the formula is

Address of a[i][j] = Base + w * (i + m * j)

where Base is base address
w is size of memory
m is row size
n is column size
i is row index
j is column index

Q. Given an array a[3][4], base = 100, w = 4 bytes, m = 3, n = 4, find address of a[1][3] in column major order.

Sol- Base = 100, w = 4 bytes, m = 3, n = 4, i = 1, j = 3

Address of a[1][3] = 100 + 4 * (1 + 3 * 3)
= 140

Q. Given an array a[4][5], base = 1000, m = 4, n = 5, w = 8, find address of a[3][1] in col. major order.

Sol- Base = 1000, w = 8 bytes, m = 4, n = 5, i = 3, j = 1

Address of a[3][1] = 1000 + 8 * (3 + 4 * 1)
= 1056

SPARSE MATRIX

→ A matrix in which majority of the memory locations are not containing any value or it contains null, then it is called sparse matrix.

Eg:

	0	1	2	3	4
0	0	0	0	0	0
1	0	10	0	0	0
2	0	0	0	0	20
3	30	0	0	0	0
4	0	0	40	50	0
5	0	60	0	0	0

a[6][5] ⇒ 30 memory
Zeros ⇒ 24
Non-zeroes → 6

	0	1	2
0	6	5	6
1	1	1	10
2	2	4	20
3	3	0	30
4	4	2	40
5	4	3	50
6	5	1	60

row size
Total no. of non-zeroes
Row index
Column index
Non-zero element

Here, we have considered an array a[6][5] that is 30 memory locations. We have stored 6 number of non-zero elements so rest 24 memory locations cont--ains zeros or null.
As we call, non-zeroes are greater than zero, so it is a sparse matrix.
When a matrix is sparse and very few non-zero elements present then we can store the matrix information using another pre-columnar matrix or triplet matrix. Above example shows matrix A[6][5] information using another matrix.

Consider a matrix a[m][n] having elements. We need to check a is sparse or not. If sparse then we need to store its information into the alternate matrix sp[nz+1][3].
Here, nz means count of non-zero.

Sparse (a[m][n], sp[nz+1][3])

Step 1: Start
Step 2: Let nz=0, i, j
Step 3: for (i=0 to m-1), incr by 1.
  Step 3.1: for (j=0 to n-1), incr by 1.
    Step 3.1.1: if (a[i][j] ≠ 0), then
      Step 3.1.1.1: nz = nz + 1
      [end of if]
    [end of for - 3.1]
  [end of for - 3]
Step 4: if (nz >= (m * n)/2)

Step 4.1: Display "it is not sparse".
Step 5: else
Step 5.1: Display "it is sparse".
Step 5.2: Alternate (a[m][n], sp[nz+1][3])
[end of if]

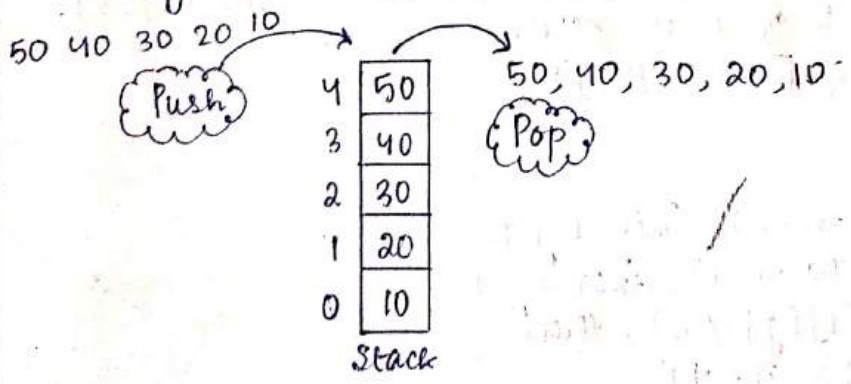
Step 6: Stop

[Alternate algorithm]
Alternate (a[m][n], sp[nz+1][3])

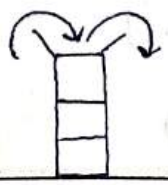
Step 1: Start
Step 2: Let i, j, k=1, sp[0][0] = m, sp[0][1] = n, sp[0][2] = nz
Step 3: for (i=0 to m-1), increase by 1
Step 3.1: for (j=0 to n-1), incr by 1.
Step 3.1.1: if (a[i][j] ≠ 0), then
Step 3.1.1.1: sp[k][0] = i
sp[k][1] = j
sp[k][2] = a[i][j]
k = k+1
[end of if]
[end of for]
[end of for]
Step 4: Stop

STACK

→ A linear data structure which implements last in first out concept (LIFO) is called stack. A stack can be implemented using array and linked list.
Eg: a) Stack of books
b) Stack using array
Eg: int a[5];



Stack using array:
Consider an array stk[5]
Eg: stack diagram



The main operation of stack are : push, pop and traverse

Push :

→ It is to insert an element into stack at top position.

Pop :

→ It is to delete an element from top position of a stack.

Traverse or display :

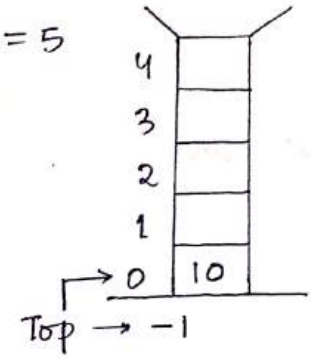
→ It is to visit (or) display all elements of a stack.

Top :

→ It is a special variable which hold the address of topmost element. Initially when the stack is empty, the top variable contains null (or) -1.

Consider an example :-

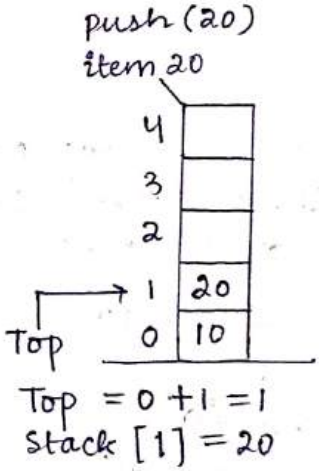
Declare an array stack [size] where size = 5
initially stack is empty so top = -1



Series of operation to execute :

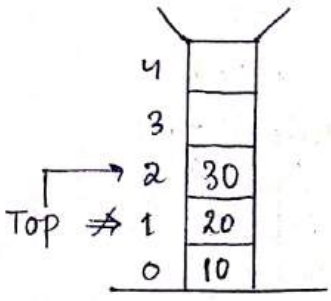
Push (10) : Here item = 10
top = 1
stack [top] = item
so, top = -1 + 1 = 0
stack [0] = item is 10

Push (20) : item = 20
top = 0 + 1 = 1
stack [i] = 20

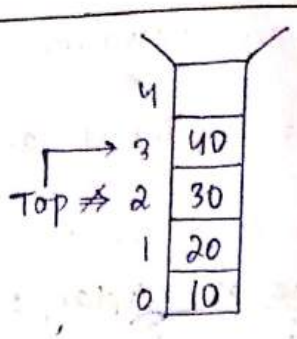


Top = 0 + 1 = 1
Stack [1] = 20

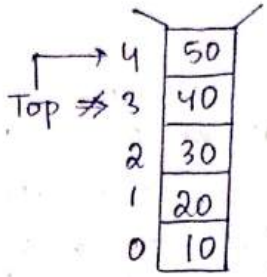
Push (30) : item = 30
top = 1 + 1 = 2
stack [2] = 30
and so on



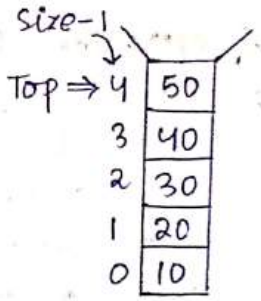
Push (40)



Push (50)



Push (60)

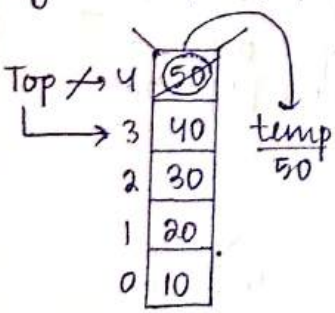


when top = size - 1 i.e. 4
The stack is full or overflow.

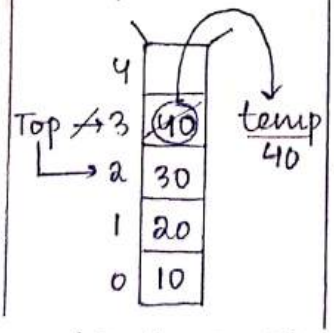
Now let us pop elements from stack [top] as :

temp = stack [top]
Top = Top - 1

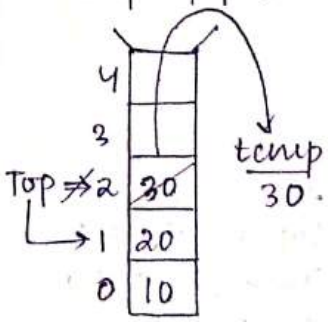
Eg : temp = pop()



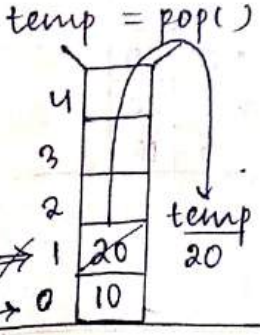
temp = pop()



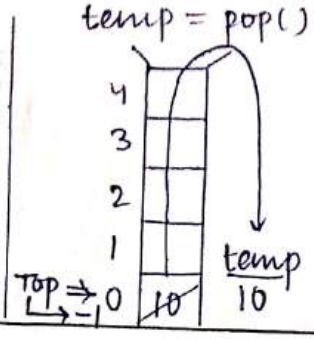
temp = pop()



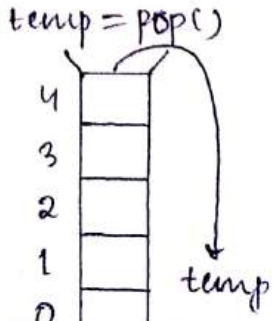
temp = pop()



temp = pop()



temp = pop()



Top → -1

when top = -1 or NULL then stack is empty or underflow.

Algorithm

Consider an array stack[size], Top represents index of topmost element. Initially Top = -1.

Push (Stack[size], top, item)
Step 1 : Start
Step 2 : if (top = size - 1) then
  Step 2.1 : Display "Stack overflow"
Step 3 : else
  Step 3.1 : Top = top + 1
  Step 3.2 : Stack[top] = item
    [end of if]
Step 4 : Stop

Pop (stack[size], top, temp)
Step 1 : Start
Step 2 : if (Top = -1) then
  Step 2.1 : Display "Stack underflow"
Step 3 : else
  Step 3.1 : temp = stack[top]
  Step 3.2 : Display "Popped is", temp
  Step 3.3 : top = top - 1
    [end of if].
Step 4 : Stop

APPLICATIONS OF STACK

→ The concept of stack can be implemented in different application areas. Few important applications are :
1) Keeping track of function calls
2) Recursive Functions
3) Conversion and evaluation of Arithmetic expressions.
4) Tree traversal
5) Graph Traversal.

Conversion and Evaluation of arithmetic expression.

The arithmetic expression can be written in 3 ways :-

1) Infix expression :
→ Here, the operator is placed in between the operands.
  Eg : a + b

2) Prefix expression :
→ Here, the operator is placed before the operands.
  Eg : +ab

3) Postfix expression :

→ Here, the operator is placed after the operands.

Eg : ab+

→ If an infix expression is given (a+b*(c-d/e)+f) evaluating this expression takes more time compared to converting this expression into postfix and then evaluating it.

General conversion procedure

They are :

1) Infix to prefix
2) Prefix to infix
3) Infix to postfix
4) Postfix to infix
5) Prefix to postfix
6) Postfix to prefix

A. Infix to prefix :

Given an infix expression : (a*b-(c+d^e)/f)

(a*b-(c+d^e)/f)
= (a*b-(c+[^de])/f)
= (a*b-([+c^de])/f)
= ([*ab]-[+c^de]/f)
= [*ab]-[/+c^def]
= [-*ab/+c^def]

Priority of Operators

Levels	Operators
1	exponent ^
2	*, /, %
3	+, -

STEPS TO SOLVE

1) Identify the innermost bracket and the operator for evaluation.
2) Convert the operator and its operands into prefix.
3) Continue this process repeatedly till the whole expression is converted.

B. Prefix to infix :

Given prefix expression :

-*ab/+c^def
= -*ab/+c[d^e]f
= -*ab/[c+d^e]f
= -*ab[(c+d^e)/f]
= -[a*b][(c+d^e)/f]
= [a*b-(c+d^e)/f]

1) Here, read each operator in right to left order for evaluation.
2) Identify the operator and its immediate next 2 operands and connect them into infix.
3) Continue this process repeatedly
4) If any of the operator within square bracket having lower priority, than the operator identif , then use ( ) for the data.

1) Here, read each operator in right to left order for evaluation.
2) Identify the operator and its immediate next 2 operands and convert them into infix.
3) Continue this process repeatedly
4) If any of the operator within square bracket having lower priority than the operator identified, then use ( ) for the data.

(Unseen) 3/7/24

C. Infix to Postfix:

Consider an infix expression given for conversion:

(a * b - (c + d ^ e) / f)
= (a * b - (c + [de^]) / f)
= (a * b - [cde^+] / f)
= ([ab*] - [cde^+] / f)
= ([ab*] - [cde^+f/])
= [ab*cde^+f/-]

PROCEDURE

1) Identify the innermost bracket and operator for evaluation.
2) Convert the operator and its operands into postfix and write within square bracket.
3) Continue this process repeatedly till whole expression is converted.

D. Postfix to Infix:

Consider postfix expression given for conversion:

ab*cde^+f/-
= [a*b]cde^+f/-
= [a*b]c[d^e]+f/-
= [a*b][c+d^e]f/-
= [a*b][(c+d^e)/f]-
= (a*b-(c+d^e)/f)

PROCEDURE

1) Identify each operator from left to right order.
2) Convert the operator and its its previous two operands from postfix to infix.
3) Continue this process till whole expression is converted to infix.

E. Prefix to Postfix

Given prefix expression:

$-*ab/+c\hat{}def$

$= -*ab/+cde\hat{}f$

$= -*ab/[cde\hat{}+f]$

$= -*ab[cde\hat{}+f/]$

$= -[ab*][cde\hat{}+f/]$

$= [ab*cde\hat{}+f/-]$

PROCEDURE

1) Read an operator from right to left order in prefix notation.
2) Convert the operator and its immediate two operands into postfix and write within square brackets [ ].
3) Continue this process repeatedly till the whole expression is converted.

F. Postfix to Prefix

Given postfix expression:

$ab*cde\hat{}+f/-$

$= [*ab]cde\hat{}+f/-$

$= [*ab]c[\hat{}de]+f/-$

$= [*ab][+c\hat{}de]f/-$

$= [*ab][/+c\hat{}def]-$

$= [-*ab/+c\hat{}def]$

PROCEDURE

1) Read each operator in left to right order from postfix notation.
2) Convert the operator and its previous 2 operands into prefix and write within [ ].
3) Continue this process till the whole expression is converted.

Conversion and Evaluation using stack

→ Stack supports 1) infix to postfix conversion
2) Postfix Evaluation
3) Infix to prefix conversion
4) Prefix Evaluation

A. Infix to Postfix conversion using Stack

Given an infix expression : Q = a+b*(c-d^e)/f

Symbol	Stack	Postfix P
	(	
a	(	a
+	(+	a
b	(+	ab
*	(+ *	ab
(	(+ * (	ab
c	(+ * (	abc
-	(+ * (-	abc
d	(+ * (-	abcd
^	(+ * (- ^	abcd
e	(+ * (- ^	abcde
)	(+ *	abcde^-
/	(+ /	abcde^-*
f	(+ /	abcde^-*f
)	EMPTY	abcde^-*f/+

Given an infix expression: Q = b*c - (d/e^f)+g/h
The postfix expression shall be contained by P. Stack is declared which holds the operators and opening brackets.

Postfix (Q, P, Stack)

Step 1: Start
Step 2: Push '(' into stack and put ')' at end of Q.
Step 3: while (stack is not empty)
  Step 3.1: scan a symbol from Q
  Step 3.2: if(Q[i] is operand) then
    Step 3.2.1: put Q[i] at end of P.
  Step 3.3: else if (Q[i] is '(' then
    Step 3.3.1: Push Q[i] into stack
  Step 3.4: else if (Q[i] is operator) then
    Step 3.4.1: while (stack[top] is operator
                AND
              priority (stack[top]) >= priority (Q[i])
      Step 3.4.1.1: pop operators from stack [top] AND store at the end of P.
    [end of while]
    Step 3.4.2: Push Q[i] into stack.
  Step 3.5: else if (Q[i] is ')') then
    Step 3.5.1: while (stack[top] is not equal to '(')
      Step 3.5.1.1: pop operators from stack [top] and store at the end of P.
    [end of while]
    Step 3.5.1.2: delete '(' from stack [top]

  [end of if -3.2]
  [end of while -3]
Step 4: Display postfix P
step 5: Stop

For example, given : Q = b*c-(d/e^f)+g/h)

Symbol (Q[i])	Stack	Postfix (P)
	(	
b	(	b
*	(*	b
c	(*	bc
-	(-	bc*
(	(-(	bc*
d	(-(	bc*d
/	(-(/	bc*d
e	(-(/	bc*de
^	(-(/^	bc*de
f	(-(/^	bc*def
)	(-	bc*def^/
+	(+	bc*def^/-
g	(+	bc*def^/-g
/	(+/	bc*def^/-g
h	(+/	bc*def^/-gh
)	EMPTY	bc*def^/-gh/+

Postfix evaluation using stack :

Given postfix expression P = BC * DEF^/ -

Let us consider some values for operands

B=5, C=6, D=10, E=2, F=7

Now let us represent the postfix P using these values

P = 5,6,*,10,2,7,^,/,-,)

Symbols	Stack	A	B	C = B ⊗ A
5	5			
6	5, 6			
*	30	6	5	C = 5 * 6 = 30
10	30, 10			
2	30, 10, 2			
7	30, 10, 2, 7			
^	30, 10, 128	7	2	2^7 = 128
/	30, 0	128	10	C = 10/128 = 0
−	30	0	30	C = 30 − 0 = 30
)	30			

Algorithm:

Postfix (P, Stack)

Step 1 : Start
Step 2 : Let C, A, B, ⊗, i
Step 3 : Store ')' at end of postfix P
Step 4 : While (P[i] ≠ ')')
  Step 4.1 : if (P[i] is operand) then
    Step 4.1.1 : Push (P[i]) into stack
  Step 4.2 : else if (P[i] is operator ⊗) then
    Step 4.2.1 : A = pop 1st element from stack [top]
    Step 4.2.2 : B = pop 2nd element from stack [top]
    Step 4.2.3 : C = B ⊗ A
    Step 4.2.4 : C = push result C into stack [top]
      [end of if]
  [end of while]
Step 5 : Display "Result is", stack [top]
Step 6 : Stop

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

Find its equivalent postfix and then evaluate it using stack.
Consider a = 10, b = 20, c = 2, e = 4, f = 5, g = 4

(a * b - (c ^ e / f) * g)
= a * b - ([ce^] / f) * g)
= a * b - ([ce^f /] * g)

= [ab *] - ([ce^f/] * g)
= [ab*] - [ce^f/g*]
= [ab*ce^f/g*-] = P

Representing postfix P using values given :-
P = 10, 20 * 2, 4 ^ 5, / 4, *, -,)

OR

Converting (a * b - (c ^ e / f) * g) into postfix P using stack -

Symbol	Stack	Postfix
	(	
a	(	a
*	(*	a
b	(*	ab
-	(-	ab *
(	(- (	ab *
c	(- (	ab * c
^	(- (^	ab * c
e	(- (^	ab * ce
/	(- (/	ab * ce^
f	(- (/	ab * ce^f
)	(-	ab * ce^f /
*	(- *	ab * ce^f /
g	(- *	ab * ce^f/g
)	EMPTY	ab * ce^f/g * -

For the given postfix
P = ab * ce^f/g * -)
Consider a=10, b=20, c=2, e=4, f=5, g=4
P = 10, 20, *2, 4^, 5, /, 4, *, -,)

Symbol	Stack	A	B	C = B ⊗ A
10	10			
20	10, 20			
*	200	20	10	C = 10 * 20 = 200
2	200, 2			
4	200, 2, 4			
^	200, 16	4	2	C = 2 ^ 4 = 16
5	200, 16, 5			
/	200, 3	5	16	C = 16/5 = 3
4	200, 3, 4			
*	200, 12	4	3	C = 3 * 4 = 12
−	188	12	200	C = 200 − 12 = 188
)	188			

Queue

→ A linear data structure which implement first in first out concept (FIFO) is called queue.

Types of Queue

There are 4 types :-

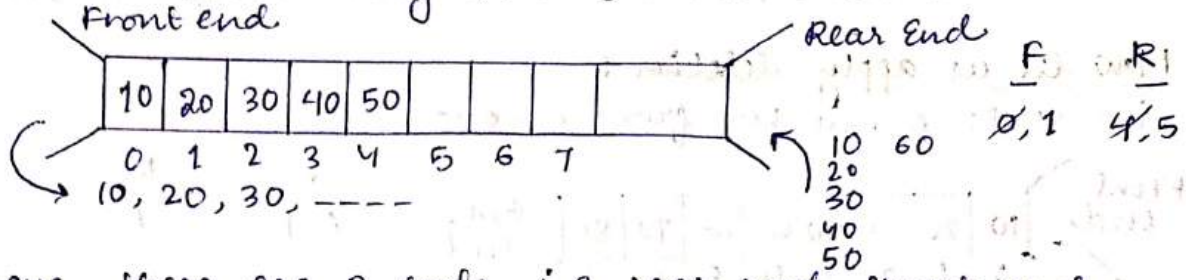
i) Linear Queue
ii) Circular Queue
iii) Double ended Queue
iv) Priority Queue

The concept of queue can be implemented using array and linked list.

Queue using array :

A. Linear Queue

Let us consider an array Q[size] where size = 8.



In Queue, there are 2 ends, i.e rear and front end. The rear end identifies insertion and front end identifies deletion. There are 2 special variables are used : front and rear. Front contains the index of first element and rear contains the index of last element.

→ The common operations on queue are :

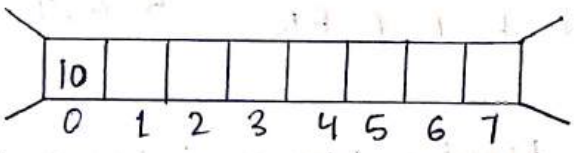
1) Insertion
2) Deletion
3) Traversion or display.

Consider an example - (For insertion).

Consider an array Q[size] where size = 8.

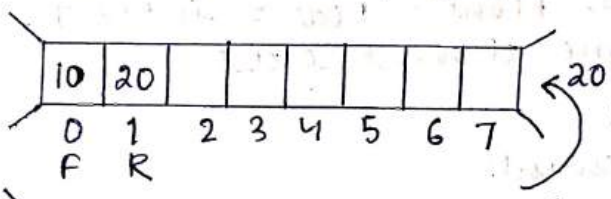
F = -1, R = -1

Insert 10 :



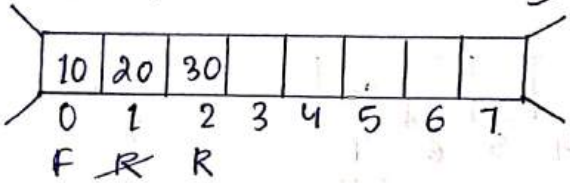
F: ~~-1~~ 0 R: ~~-1~~ 0

Insert 20 :



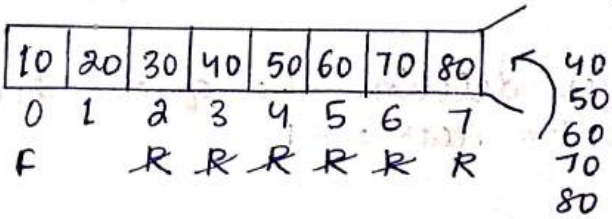
F: 0 R: Ø, 1

Insert 30 :



F: 0 R: ~~1~~ 2

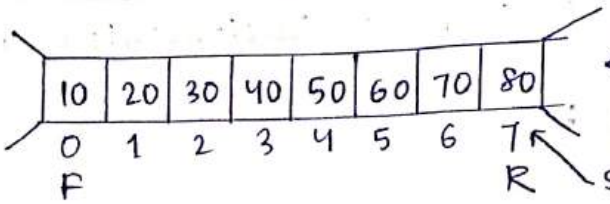
Insert 40, 50, 60, 70, 80 :



F: 0 R: ~~2~~, 7

....

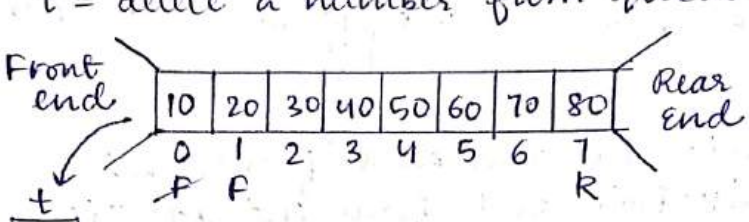
Insert 90:



when rear = size - 1, it indicates queue is full or overflow.

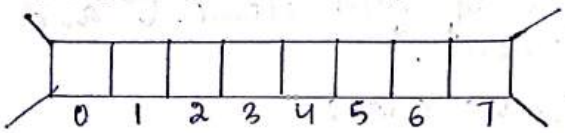
Now let us apply deletion:

t = delete a number from queue.

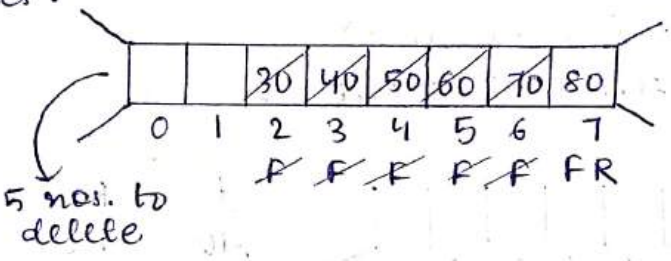


F: 0, 1 R: 7

t = delete a number



Similarly during deletion the front will move forward to the next indices. Here, if we delete 5 nos. then the queue becomes:

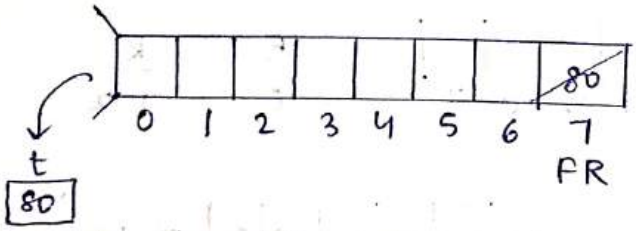


F: ~~2, 3, 4, 5, 6, 7~~ R: 7

Now, we can observe Front = Rear = index 7.
So, there is only one element exist.

Now, if we delete:

t = delete an element



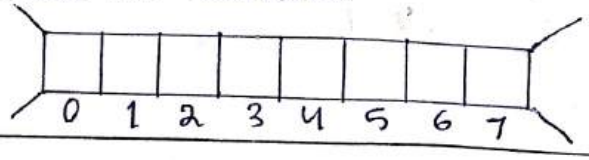
F: ~~7~~ -1 R: ~~7~~ -1

Here after deletion the queue becomes empty.
So, front = Rear = -1

Now, if we execute

t = delete an element

F: -1 R: -1



It shows queue is empty or underflow.

Algorithm (Insertion)

Consider a queue Q[size], Front, Rear.
Initially the queue is empty and Front = -1 and rear = -1.
Item is a variable which holds a value for insertion.

Insertion (Q[size], Front, Rear, Item)
Step 1 : Start
Step 2 : if (Rear = size - 1) then
  Step 2.1 : Display "Queue is full or overflow"
  Step 2.2 : Exit
Step 3 : else if (Front = -1 AND Rear = -1) then
  Step 3.1 : Front = Rear = 0,
  Step 3.2 : Q[Rear] = item
Step 4 : else
  Step 4.1 : Rear = Rear + 1
  Step 4.2 : Q[Rear] = item
    [end of if]
Step 5 : stop

Deletion Algorithm

Deletion (Q[size], Front, Rear)
Step 1 : Start
Step 2 : if (Front = -1 AND Rear = -1) then
  Step 2.1 : Display "Queue is empty or underflow"
  Step 2.2 : Exit
Step 3 : Else if (Front = Rear) then
  Step 3.1 : Display "Deleted is", Q[Front]
  Step 3.2 : Front = Rear = -1
Step 4 : else
  Step 4.1 : Display "Deleted is", Q[Front]
  Step 4.2 : Front = Front + 1
    [end of if]
step 5 : stop

B. Circular Queue

→ The drawback of linear queue is once rear reaches to size − 1 then the queue is full. In this case even if there is a vacancy at front end still we cannot use the memory. It is because we are confined to insert at rear end only.

→ This drawback can be overcome in circular queue. In case of circular queue there is no front-end and rear-end. The array is in circular manner. Only the front and rear variables are used to identify the 1st and last elements indices. Here insertion & deletion will be executed in circular manner.

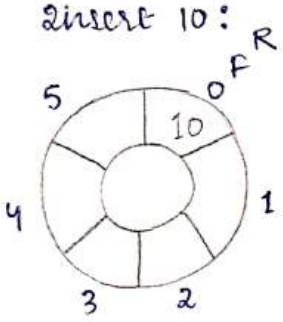
Example :

Consider a Queue Q[size] where size = 6

Initially Front = −1 and Rear = −1.

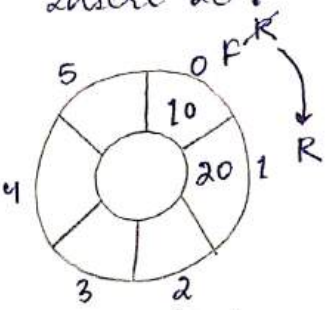
$R = (R+1) \% size$

Insert 10:



F	R
~~−1~~, 0	~~−1~~, 0

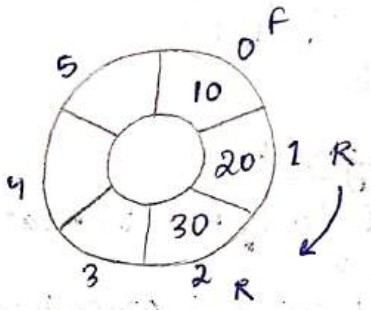
Insert 20:



F	R
0	~~0~~, 1

$R = (0+1) \% 6$
$= 1$

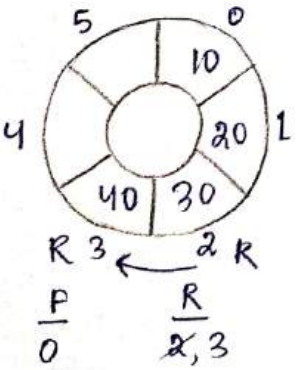
Insert 30:



F	R
0	~~1~~, 2

$R = (1+1) \% 6$
$= 2 \% 6$
$= 2$

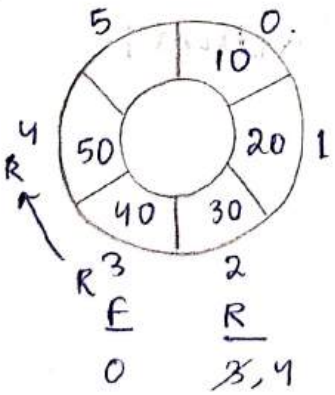
Insert 40:



F	R
0	~~2~~, 3

$R = (2+1) \% 6$
$= 3$

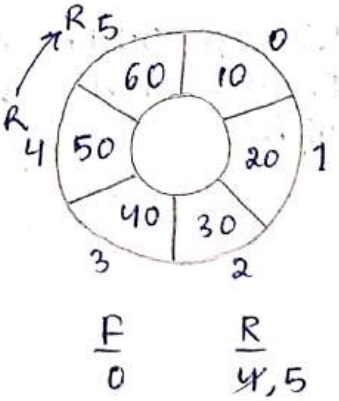
Insert 50:



F	R
0	~~3~~, 4

$R = (3+1) \% 6$
$= 4 \% 6$
$= 4$

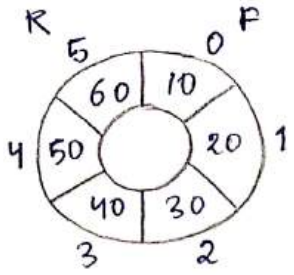
Insert 60:



F	R
0	~~4~~, 5

$R = (4+1) \% 6$
$= 5$

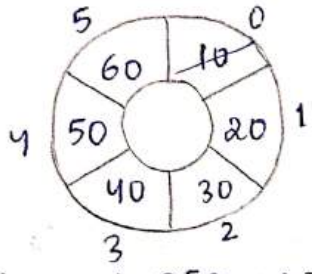
Insert 70:



F	R
0	5

Queue is full or overflow

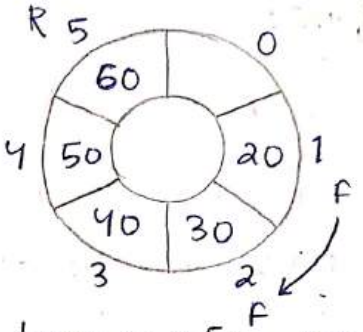
Delete a number into temp



temp ⇒ Q[front] ⇒ 10

F	R
~~0~~, 1	5

F = (F+1) % size
= (0+1) % 6 = 1

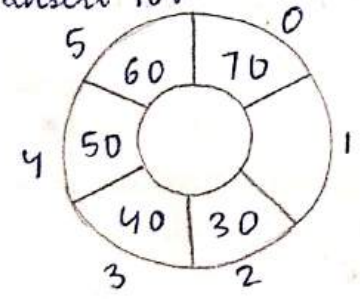


temp = Q[Front] ⇒ 20

F	R
~~1~~, 2	5

F = (1+1) % 6 = 2

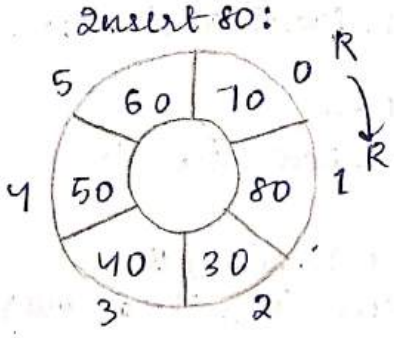
Insert 70:



F	R
2	5

R = (R+1) % size
= (5+1) % 6 = 0

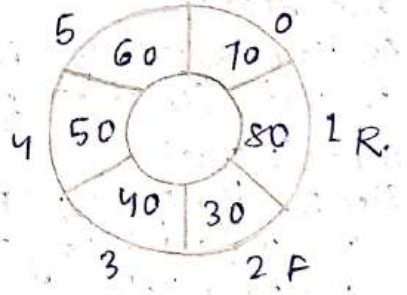
Insert 80:



F	R
2	0

R = (0+1) % 6
= 1

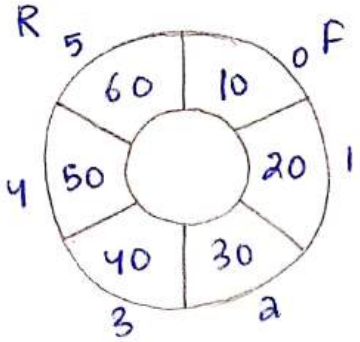
Insert 90:



f	= (R+1) % size
2	= (1+1) % 6
2	= 2

⇒ Queue is full or overflow.

Another scenario :



F = (R+1) % size
0 = (5+1) % 6
0 = 0
→ Queue is full or overflow

Queue Overflow :

Situation 1 :
When Front = 0 AND rear = size - 1 then Queue is full
Situation 2 :
When front = rear + 1 then Queue is full
Both situations 1 & 2 can be dealt using a single formula :
Front = (rear + 1) % size then Queue is full.

Queue Underflow :

When Front = -1 AND rear = -1 then Queue is Empty

Algorithm :

Consider a queue Q[size], front and rear. Initially front = -1 and Rear = -1
An item given for insertion
insertion (Q[size], front, rear, item)

Step 1 : start
Step 2 : if (front = (Rear + 1) % size) then
  Step 2.1 : Display "Queue is full or overflow"
  Step 2.2 : Exit
Step 3 : else if (front = -1 AND rear = -1) then
  Step 3.1 : Front = Rear = 0
  Step 3.2 : Q[Rear] = item
Step 4 : Else
  Step 4.1 : Rear = (Rear + 1) % size
  Step 4.2 : Q[Rear] = item
    [end of if]
Step 5 : Stop

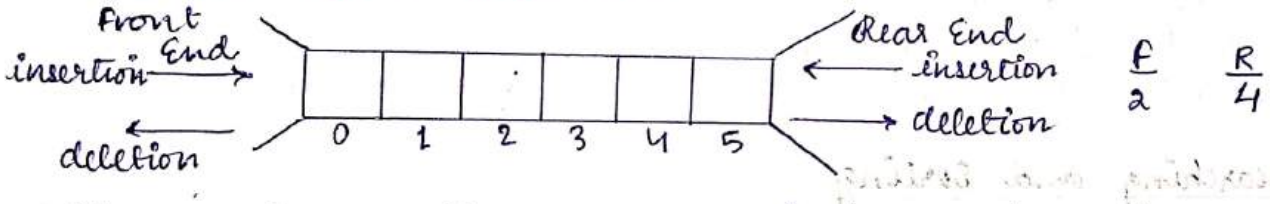
Algorithm:

Deletion (Q[size], Front, Rear)
Step 1 : Start
Step 2 : if (front = -1 AND rear = -1) then
  Step 2.1 : Display "Queue is Empty or underflow"
  Step 2.2 : exit
Step 3 : else if (front = rear) then
  Step 3.1 : Display "deleted", Q[Front]
  Step 3.2 : Front = Rear = -1
Step 4 : else
  Step 4.1 : Display "deleted", Q[Front]
  Step 4.2 : Front = (Front + 1) % size
    [end of if]
Step 5 : Stop

C. Double Ended Queue

→ It is a linear queue in which insertion and deletion is possible at both ends such as:

1) Insertion at Rear End
2) Deletion from front end
3) Insertion at frontend
4) Deletion from rear end



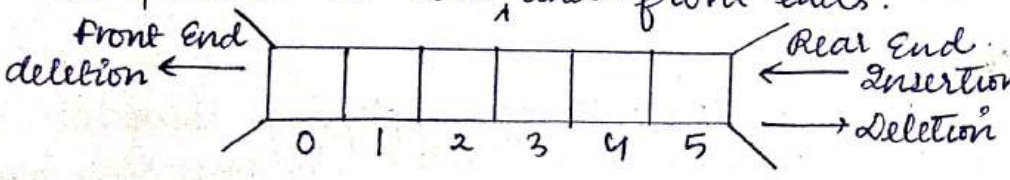
→ If we apply all the four operations, then it may not follow FIFO concept.

→ A double ended queue can be used in 2 ways:

1) Input restricted Deque
2) Output restricted Deque.

Input restricted Deque

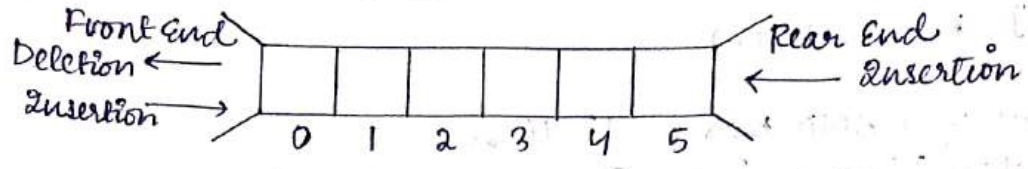
→ Here insertion is done only at rear end but the deletion is possible at both rear and front ends.



Output restricted Deque

→ In this case, the deletion is restricted to one end, it means deletion is possible only at front end but insertion is

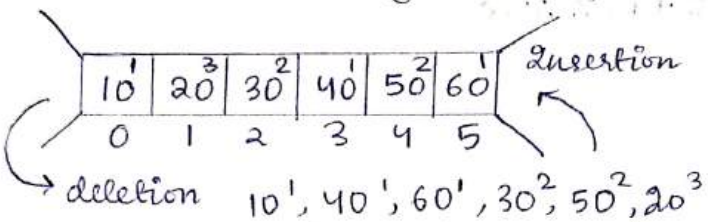
possible at both ends.



D. Priority Queue

→ It is a linear queue in which each element is inserted along with its priority level. Here, the deletion is performed based on the priority level serially.

→ The priority queue can also be used as ascending priority queue and descending priority queue.



$10^1, 40^1, 60^1, 30^2, 50^2, 20^3$

Numbers	Priority
10	1
20	3
30	2
40	1
50	2
60	1

Applications of Priority Queue

→ Print queue maintained by the print server
→ Online ticket reservation
→ E-mail system
→ Graph: traversal method
→ Round robin technique in operating system

Searching and Sorting

Searching:

→ The common 2 searching methods are:
1) Linear Search
2) Binary Search

A. Linear Search

→ When we search for an item serially one after the other from lower bound to upper bound, then it is called linear search.

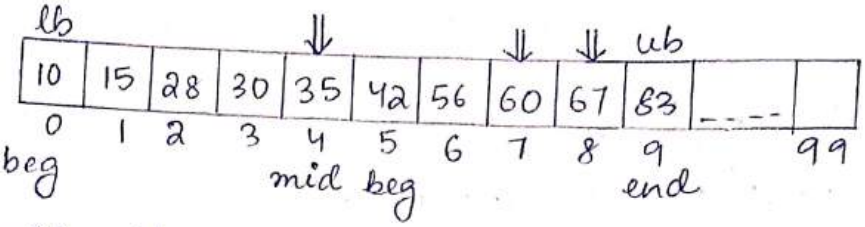
B. Binary Search

→ When we search for an element by applying divide and conquer method then it is called binary search. The binary search is applicable only when the elements are in sorted order.

Example:

Consider an example a[100] having elements from lb to ub in sorted manner.

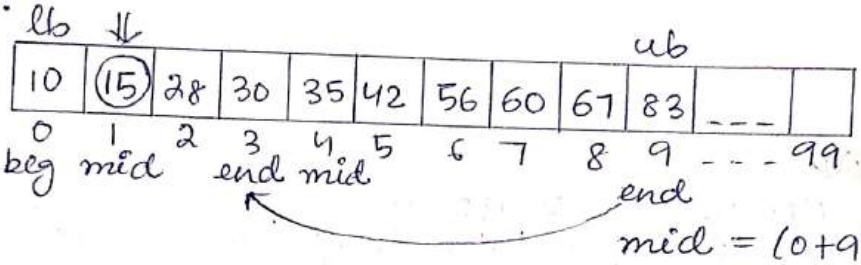
item: 67



mid = (beg + end)/2 = (0+9)/2 = 4
mid = (5+9)/2 = 7
mid = (8+9)/2 = 8

Example 2:

item: 15



mid = (0+9)/2 = 4
mid = (0+3)/2 = 1

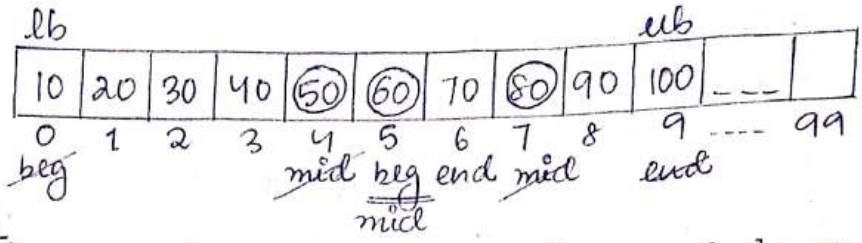
Algorithm

Given an array a[100] with elements from lb to ub an item given to search.

Binary search (a[100], lb, ub)
Step 1: Start
Step 2: let beg, end, mid
Step 3: beg = lb, end = ub, mid = (beg + end)/2
Step 4: while (beg <= end AND a[mid] ≠ item)
  Step 4.1: if (item < a[mid]) then
    Step 4.1.1: end = mid - 1
      [end of if]
  Step 4.2: if (item > a[mid] then
    Step 4.2.1: beg = mid + 1
      [end of if]
  Step 4.3: mid = (beg + end)/2
    [end of while]
Step 5: if (a[mid] = item) then
  Step 5.1: Display "Item found at location", mid
Step 6: else
  Step 6.1: Display "Item not found"
    [end of if]
Step 7: Stop

Example 3:

Item: 60



Sorting:

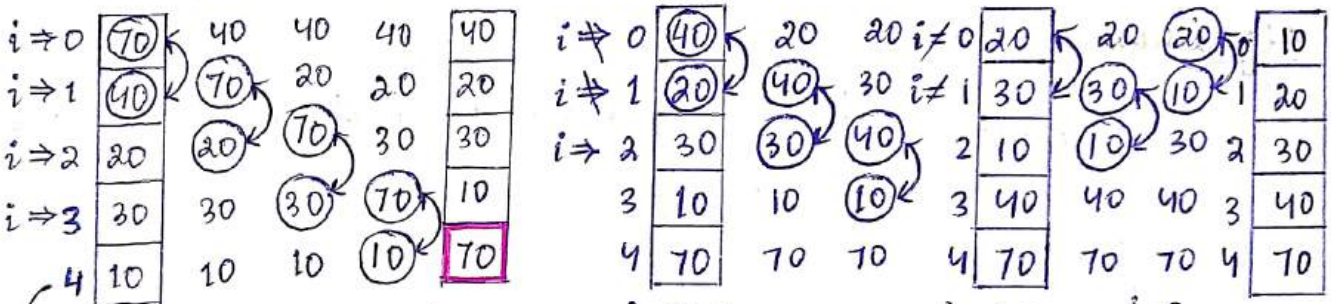
→ Arranging the elements either in ascending or descending or in some order is called sorting.

few important sorting methods are:

1) Selection Sort
2) Bubble Sort
3) Insertion Sort
4) Radix Sort
5) Quick Sort

A. Bubble sort

→ Here, each element is compared with its adjacent element, if the element is bigger then swapping is applied.



size-1 i=0,1,2,3 → size-1-k (4, 1)

if (a[i] > a[i+1]) then swap

or ub-k

In bubble sort, the larger element will flow down by bubbling up to the end.

```
for (k=1, 2, 3, 4)
{
 for (i=0
 {
  if (a[i] > a[i+1])
    swap
```

Algorithm (Bubble sort)

Bubble sort (a[size], lb, ub, temp, i, k)

Step 1: Start

Step 2: for (k=1 to ub), incr by 1.

Step 2.1: for (i=lb to ub-k), incr by 1.

Step 2.1.1: if (a[i] > a[i+1] then

Step 2.1.1.1: temp = a[i]

Step 2.1.1.2: a[i] = a[i+1]

Step 2.1.1.3: a[i+1] = temp

[end of if]

[end of for]

[end of for]

Step 3: Stop

→ In bubble sort, the larger elements will go down by bubbling up the smaller elements.

B. Insertion sort

→ In this sorting method, we always insert an element by finding the appropriate location in a sorted list.

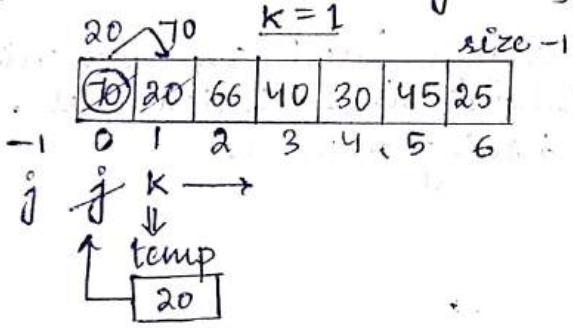
Eg: Consider an array a[7] having elements.

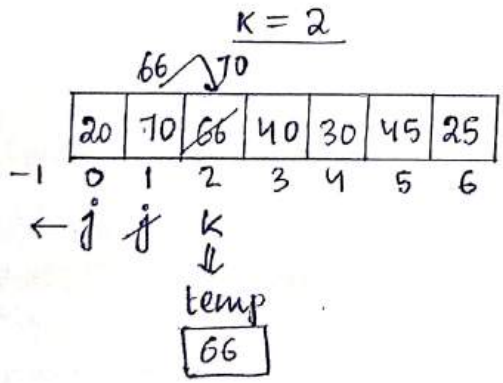
Algorithm (Insertion Sort)

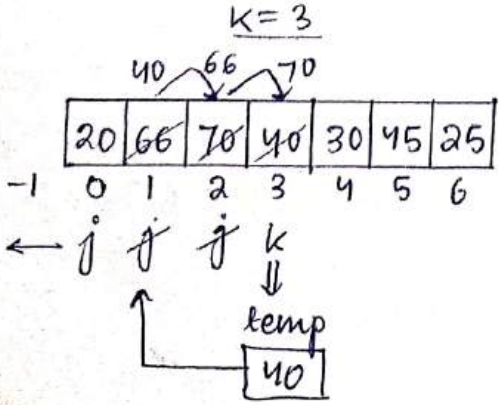
Consider an array a[size], size = 7
Insertion sort(a[size])

Step 1 : Start
Step 2 : let j, k, temp
Step 3 : for (k = 1 to size − 1), incr by 1
  Step 3.1 : j = k − 1
  Step 3.2 : temp = a[k]
  Step 3.3 : while (j >= 0 AND a[j] > temp)
    Step 3.3.1 : a[j + 1] = a[j]
    Step 3.3.2 : j = j − 1
        [end of while]
  Step 3.4 : a[j+1] = temp
      [end of for]
Step 4 : Stop

Eg : 1 - Consider an array a[7]







C. Radix Sort

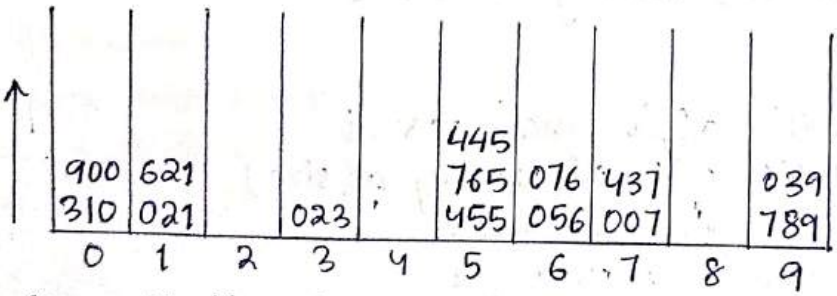
→ Here, the sorting is on digits in individual positions such as : unit position, 10's position, 100's position etc. The digits are between 0 to 9, so the base or radix is 10. That is why the sorting is called radix sort.
Here we need 10 buckets representing digits from 0 to 9.

→ Each time we need to read digits from a position and store the numbers into the respective buckets as per digit.

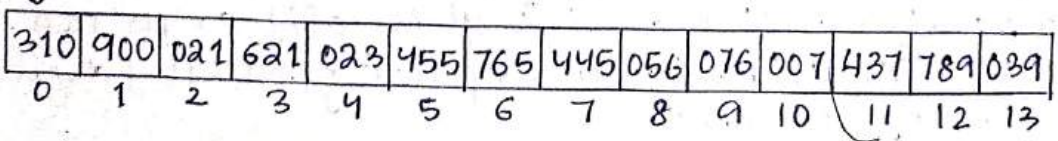
Eg: Consider an array a[14] having elements :
021, 310, 455, 789, 056, 039, 007, 900, 765, 437, 621, 445, 076, 023.

Identify largest element $l = 900$, and count no. of digits in l i.e, digits = 3
Let us consider pass = 1 to digit
means when pass = 1 ⇒ unit position
pass = 2 ⇒ 10's position
pass = 3 ⇒ 100's position

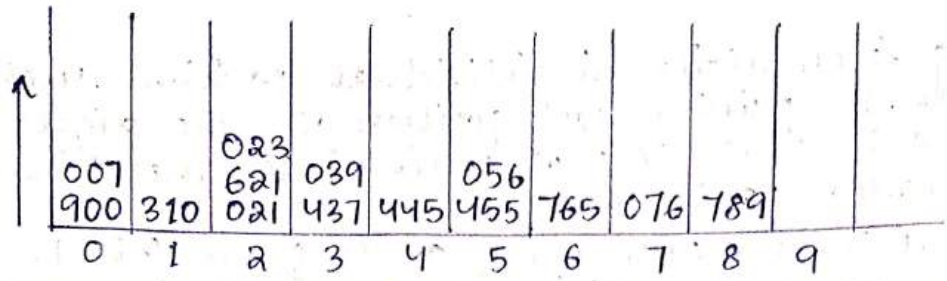
Pass 1: Initialize all the buckets from 0 to 9.
Read each element x from index 0 to 13.
Identify the unit position digit of x.
Store the element x into the corresponding bucket digit



Restore all the elements from bucket 0 to 9 serially into array :



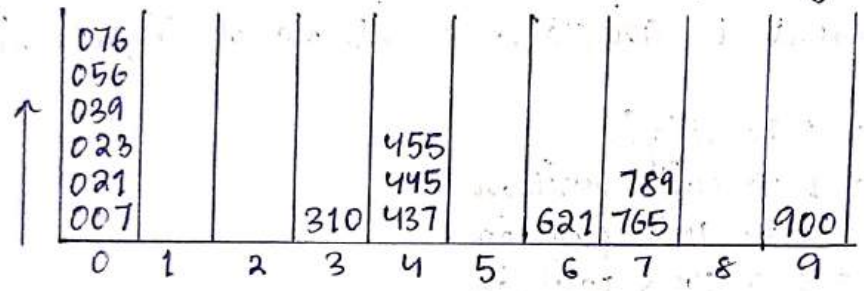
Pass 2: Read each element x from index 0 to 13.
Identify 10's position digit
Store each element x into corresponding bucket of digit



Restore all the elements from bucket 0 to 9 serially into array:

900	007	310	021	621	023	437	039	445	455	056	765	076	789
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pass 3: Read each element x from 0 to 13.
Identify 100's position digit of each x
Store each element x into corresponding bucket of digits



Restore all the elements from bucket 0 to 9 serially into array:

007	021	023	039	056	076	310	437	445	455	621	765	789	900
0	1	2	3	4	5	6	7	8	9	10	11	12	13

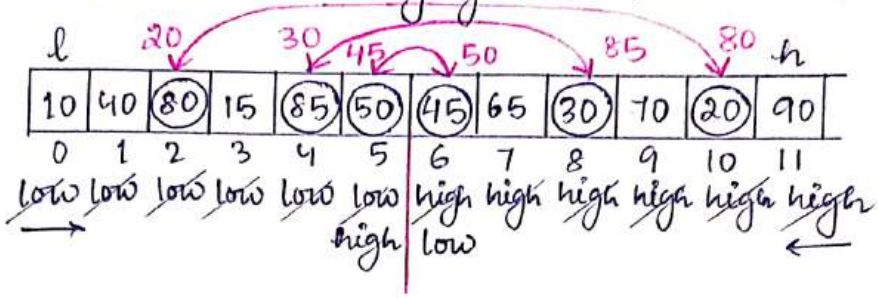
Consider an array a[size] having elements
Radix Sort (a[size])
Step 1: Start
Step 2: Let bucket (0 to 9), digit, max, pass, i
Step 3: max = largest element in array a[size]
Step 4: digit = count of digits in max
Step 5: for (pass = 1 to digit), incr by 1.
  Step 5.1: Initialize all the buckets (0 to 9)
  Step 5.2: for (i = 0 to size - 1), incr by 1.
    Step 5.2.1: digit = find a digit from a[i] as per pass.
    Step 5.2.2: Store a[i] into corresponding bucket [digit]
      [end of for - 5.2]
  Step 5.3: Restore elements from bucket (0 to 9) into array a[size]
    [end of for - 5]
Step 6: Stop

D. Quick Sort

→ It is a sorting method in which we apply divide and conqu approach. Everytime we need to choose a key value and based on the key value we need to divide the whole list of elements into 2 sub-lists.

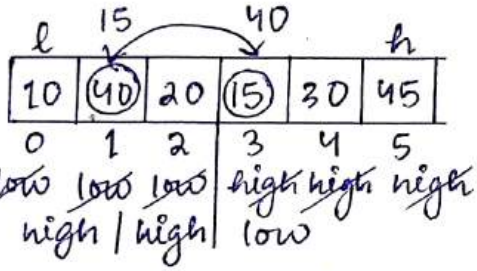
→ The left sub-list shall contain elements less than key value. The right sub-list shall contain elements greater than key value.

Eg: Consider an array given below



key = a[(l+h)/2]
= a[(0+11)/2]
= a[5]
= 50

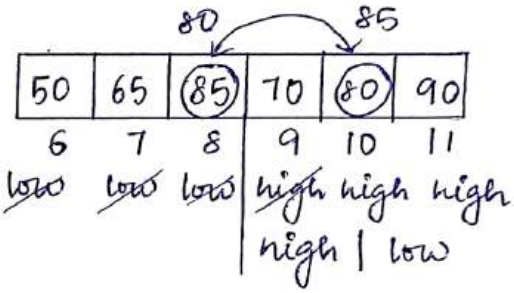
Quick [a[], l, high)



key = [(0+5)/2]
= 2
a[2] = 20

key
20

Quick (a[], low, h)



key
85

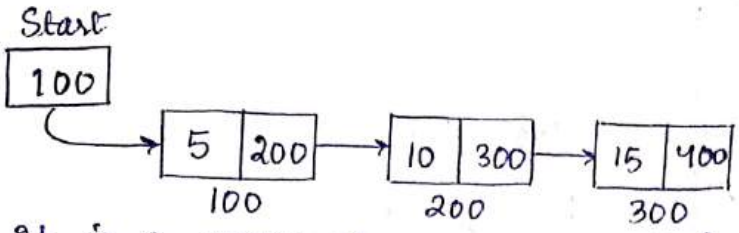
Algorithm

Quick sort (a[size], l, h)

Step 1: Start
Step 2: let low, high, key = a[(a+h)/2], temp
Step 3: while (low <= high)
  Step 3.1: while (a[low] < key)
    Step 3.1.1: low = low + 1
    [end of while]
  Step 3.2: while (a[high] > key)
    Step 3.2.1: high = high - 1
    [end of while]
  Step 3.3: if (low <= high) then
    Step 3.3.1: temp = a[low]
    Step 3.3.2: a[low] = a[high]
    Step 3.3.3: a[high] = temp
    Step 3.3.4: low = low + 1
    Step 3.3.5: high = high - 1

[end of if]
[end of while - 3]
Step 4: if(l < high) then
Step 4.1: Quick sort (a[size], l, high)
[end of if]
Step 5: if (low < h) then
Step 5.1: Quick sort (a[size], low, h)
[end of if]
Step 6: Stop

UNIT - 03 LINKED LIST



→ It is a group of non-consecutive memory locations connected in serial manner and allows operations serially is called linked list.

→ It is a linear data structure.

TYPES OF LINKED LIST

1) Single linked list
2) Circular linked list
3) Double linked list
4) Circular double linked list

A. Single linked list

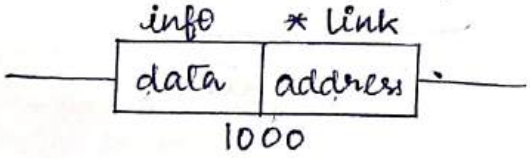
→ It is a collection of nodes.

NODE:

→ It is a memory having 2 parts:

a) **Info part:**
→ It holds the data for storage

b) **Link part:**
→ It holds the address of next node.

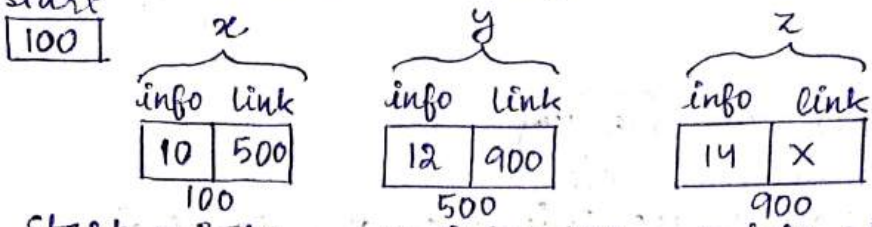


→ In C language, to create a node we need self referential structure.

→ When we define a structure in which one of the member is a pointer and it is of its own datatype; then it is called self referential structure.

Eg:
```
struct node
{
  int info;
  struct node *link;
};
```

Struct node x, y, z ;
struct node * start ;



start = &x;
x.info = 10;
x.link = &y

y.info = 12;
y.link = &z

y.info = 14;
y.link = &x.

→ It is a simple way of creating linked list with 3 nodes.

Linked list with Dynamic Memory Location :

→ We shall create a linked list with variable length using dynamic memory allocation. In C language, the memory allocation functions available are –
malloc(), calloc(), realloc(), and free()

→ We use malloc() to allocate memory
free() to release memory.

malloc()

→ It allocates the memory in bytes and returns the base address. If it cannot allocate then it returns NULL or invalid address.

SYNTAX :

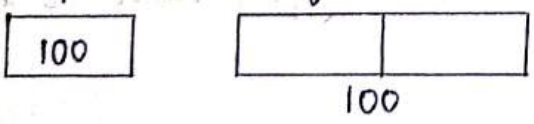
pointer variable = (struct node *) malloc (size of (struct node));

(struct node *) — cast type
size of (struct node) — no. of bytes

Eg: 1 struct node *p;

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

8 bytes



Here, if P controls NULL then it means memory allocation is failed.

If P contains address eg: 100
using P we can refer the members of node as :

p → info OR (*p).info
p → link OR (*p).link

free()

→ It is used to release the memory.

SYNTAX:

```
free (pointer);
```

→ The common operations on single linked list are :-

1) Creation
2) Insertion at the beginning
3) Insertion at the end
4) Insertion at specific location
5) Deletion of a node from the beginning
6) Deletion of a node from the end.
7) Deletion of a node from specific location.
8) Search for a node
9) Concatenating of two linked list
10) Reversing a linked list
11) Sorting
12) Merging
13) Display or traversing

→ Creating a simple linked list with 5 nodes by inserting at the beginning for that let us write C program.

```
#include <stdio.h>
#include <alloc.h>
struct node
{ int info;
  struct node *link;
};
struct node *start;
int main()
{ int n, i, start = NULL;
  struct node *fresh, *ptr;
printf("How many node to create");
scanf("%d", &n);
for (i=1; i<=n; i++)
{
  fresh = (struct node *) malloc (size of (struct node));
  if (fresh == NULL)
  {
    printf("Memory is full");
    return 0;
```

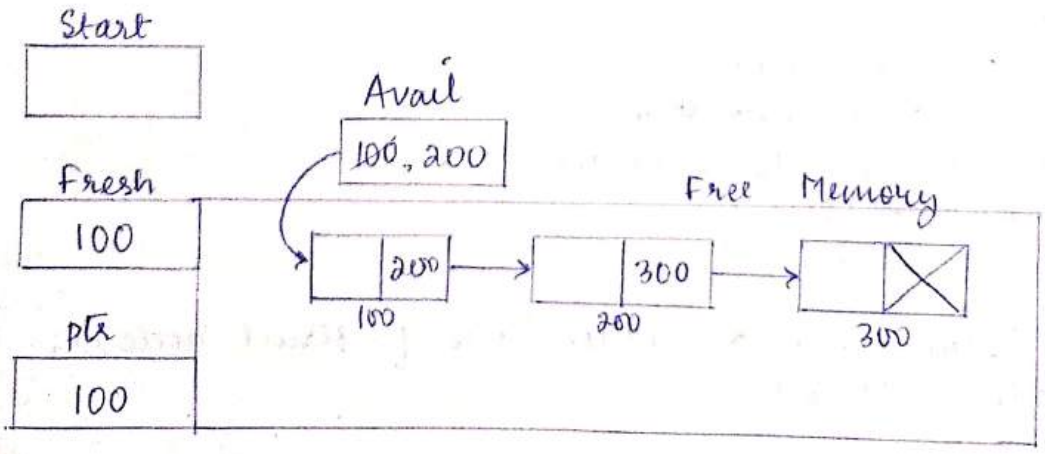
```
    }
printf ("Enter info");
scanf ("%d", &fresh → info);
fresh → link = NULL;
  if (start == NULL)
          start = Fresh;
else
   {
     fresh → link = start;
     start = fresh;
   }
}
  for (ptr = start; ptr != NULL; ptr = ptr → link)
{
  printf ("%d", ptr → info);
}
  return 0;
}
```

Let us consider two special pointers.

* start : It is used to hold address of 1st node
* fresh : It holds address of newly allocated node.

Initially start = NULL

Let us consider a free memory in RAM is used for allocation.



Algorithm (malloc)

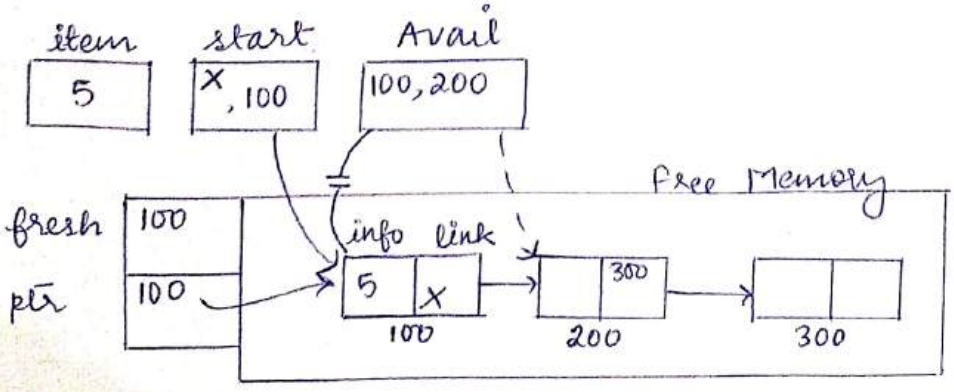
malloc()

Step 1 : Start
Step 2 : let ptr
Step 3 : if (Avail = NULL) then
  Step 3.1 : return NULL
Step 4 : else
  Step 4.1 : ptr = Avail
  Step 4.2 : Avail = link [Avail]
  Step 4.3 : return ptr
    [end of if]
Step 5 : Stop

Creation

→ It is used to create the first node of the linked list. Here, we use malloc() method to allocate memory.
  * start = NULL initially

Creation (start, fresh)
Step 1 : Start
Step 2 : let item
Step 3 : fresh = malloc()
Step 4 : if (fresh = NULL) then
  Step 4.1 : Display "Memory full"
Step 5 : else
  Step 5.1 : Display "Enter an item"
  Step 5.2 : Input item
  Step 5.3 : info [fresh] = item
  Step 5.4 : link [fresh] = NULL
  Step 5.5 : if (start = NULL) then
    Step 5.5.1 : start = fresh
      [end of if - 5.5]
  [end of if - 4]
Step 6 : Stop

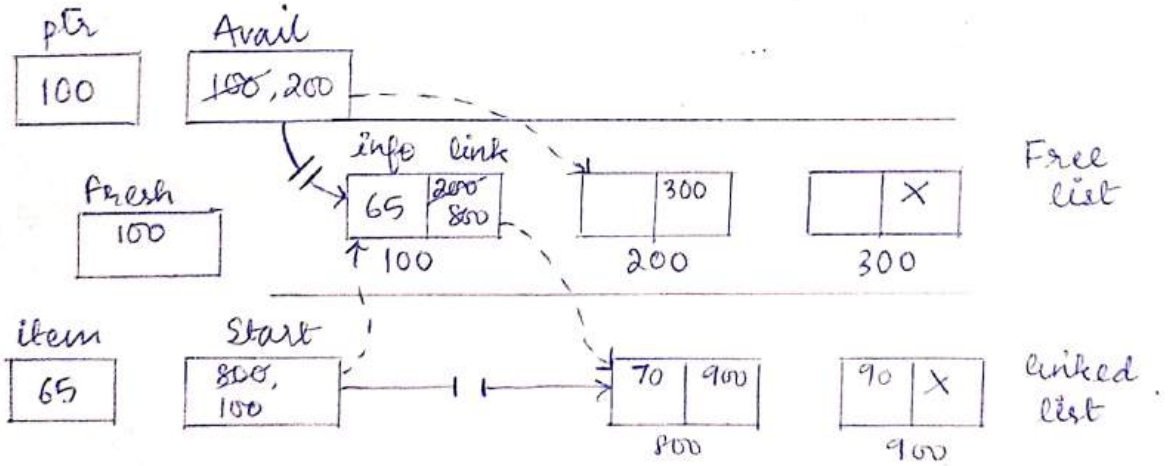


Insertion at the beginning:

[consider *start points to 1st node, *fresh used for holding new node address]

Insertion_at_first(start)

Step 1 : Start
Step 2 : let item, fresh
Step 3 : fresh = malloc()
Step 4 : if(fresh = NULL) then
  Step 4.1 : display "memory is full"
Step 5 : else
  Step 5.1 : display "enter an item"
  Step 5.2 : input item
  Step 5.3 : info[fresh] = item
  Step 5.4 : link[fresh] = NULL
  Step 5.5 : if (start = NULL) then
    Step 5.5.1 : start = fresh
  Step 5.6 : else
    Step 5.6.1 : link[fresh] = start
    Step 5.6.2 : start = fresh
      [end of if - 5.5]
  [end of if - 4]
Step 6 : Stop

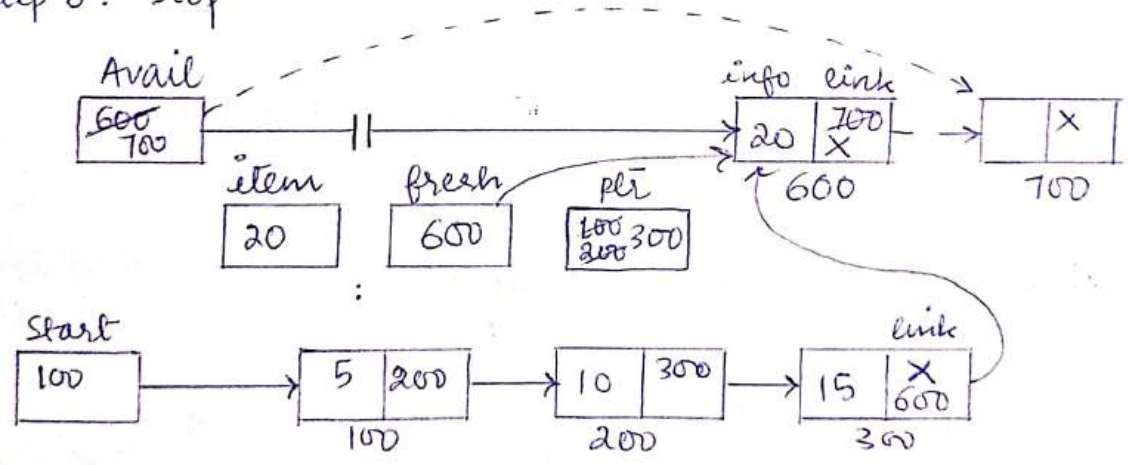


Insertion at the end :

[Consider *start points to 1st node , * fresh used for holding new node address]

Insertion_at_last (start)

Step 1 : Start
Step 2 : let item, fresh, ptr
Step 3 : fresh = malloc ()
Step 4 : if (fresh = NULL) then
  Step 4.1 : display "memory is full"
Step 5 : else
  Step 5.1 : display "enter an item"
  Step 5.2 : input item
  Step 5.3 : info [fresh] = item
  Step 5.4 : link [fresh] = NULL
  Step 5.5 : if (start = NULL) then
    Step 5.5.1 : start = fresh
  Step 5.6 : else
    Step 5.6.1 : ptr = start
    Step 5.6.2 : while (link [ptr] != NULL)
      Step 5.6.2.1 : ptr = link [ptr]
        [end of while]
    Step 5.6.3 : link [ptr] = fresh
      [end of if - 5.5]
  [end of if - 4]
Step 6 : Stop



OR

Insertion at the end:

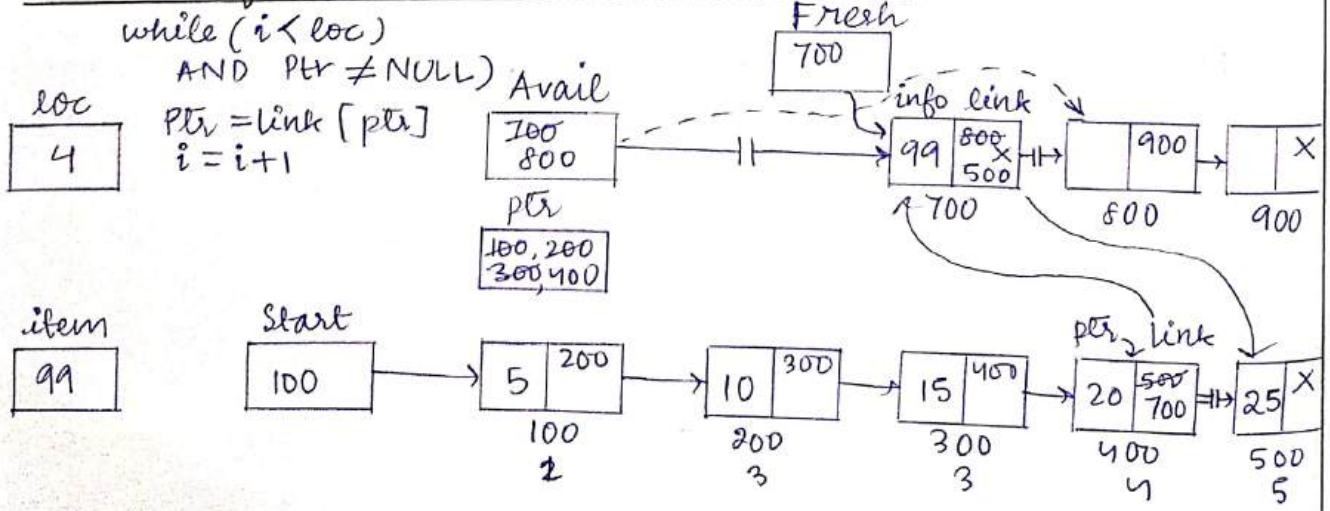
[consider *start points to 1st node, *end points to last node, *fresh used for holding new node address, item is a variable which holds a new value]

Insertion_at_last (start, end, item)

Step 1: Start
Step 2: let fresh
step 3: fresh = malloc()
Step 4: if (fresh = NULL) then
  Step 4.1: display "memory is full"
  Step 4.2: exit
    [end of if]
Step 5: info [fresh] = item
Step 6: link [fresh] = NULL
Step 7: if (start = NULL) then
  Step 7.1: start = fresh
  Step 7.2: end = fresh
Step 8: else
  Step 8.1: link [end] = fresh
  Step 8.2: end = fresh
    [end of if - 7]
Step 9: stop

* Insertion in between the nodes is possible in 2 ways:
  - Insertion after a location
  - Insertion after an item

Insertion after a location or node number:

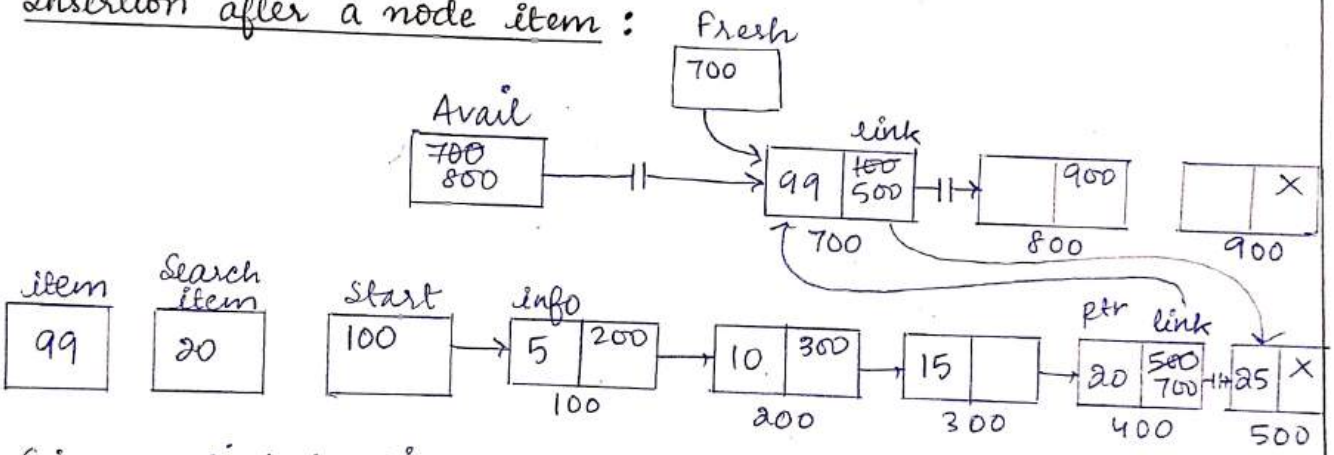


Given a linked list where start points 1st no, item and loc are given.

insertafterlocation (start, fresh, item, loc)

Step 1 : Start
Step 2 : Let i, ptr
Step 3 : fresh = malloc( )
Step 4: info [fresh] = item
Step 5: link [fresh] = NULL
Step 6: i=1, ptr = start
Step 7: while (i< loc AND ptr ≠ NULL)
  Step 7.1: ptr = link [ptr] , i = i+1
  [end of while]
Step 8 : link [fresh] = link [ptr]
Step 9: link [ptr] = fresh
Step 10 : Stop

Insertion after a node item :



Given a linked list when start points 1st node,
Given search item after which an item to be inserted

Inserted after item (Start , item, searchitem , ptr , fresh)

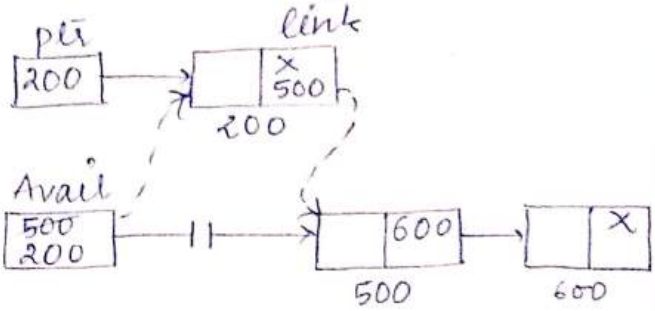
Step 1 : Start
Step 2 : fresh = malloc( )
Step 3 : info [fresh] = item
Step 4: link [fresh] = NULL
Step 5: ptr = start
Step 6: while (searchitem ≠ info [ptr] AND ptr ≠ NULL)
  Step 6.1 : Ptr = link [ptr]
  [end of while]
Step 7: if (ptr = NULL) then
  Step 7.1 : Display "Search item not found"
Step 8: else
  Step 8.1 : link [fresh] = link [ptr]

Step 8.2 : link[ptr] = fresh
[end of if]
Step 9 : stop

5) Deletion of a Node :

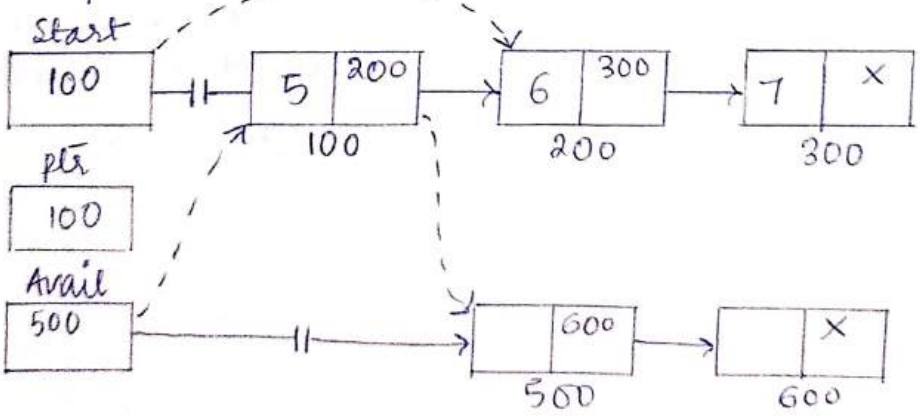
→ When we delete a node actually we return that node to the pre-memory list or avail list.
→ For returning back the node, we need 3 function

```
Frec (ptr)
Step 1 : Start
Step 2 : link[ptr] = Avail
Step 3 : Avail = ptr
Step 4 : stop
```



Deletion of 1st node :

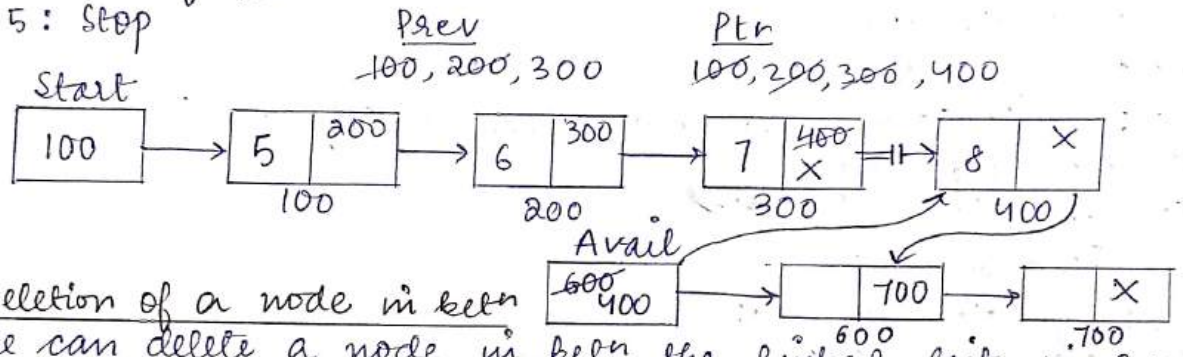
```
Deletion 1st (start, ptr)
Step 1 : Start
Step 2 : if (start = NULL), then
  Step 2.1 : Display "linked list not exist"
Step 3 : else
  Step 3.1 : ptr = start
  Step 3.2 : start = link[start]
  Step 3.3 : free (ptr)
     [end of if]
Step 4 : Stop
```



6) Deletion of last node:

deletionlastnode (start)

Step 1: Start
Step 2: let ptr, prev
Step 3: if (start = NULL), then
  Step 3.1: Display "linked list not exist"
Step 4: else
  Step 4.1: ptr = start
  Step 4.2: while (link [ptr] ≠ NULL)
    Step 4.2.1: prev = ptr
    Step 4.2.2: ptr = link [ptr]
      [end of while]
  Step 4.3: link [ptr] = NULL
  Step 4.4: free (ptr)
    [end of if]
Step 5: Stop



7) Deletion of a node in between

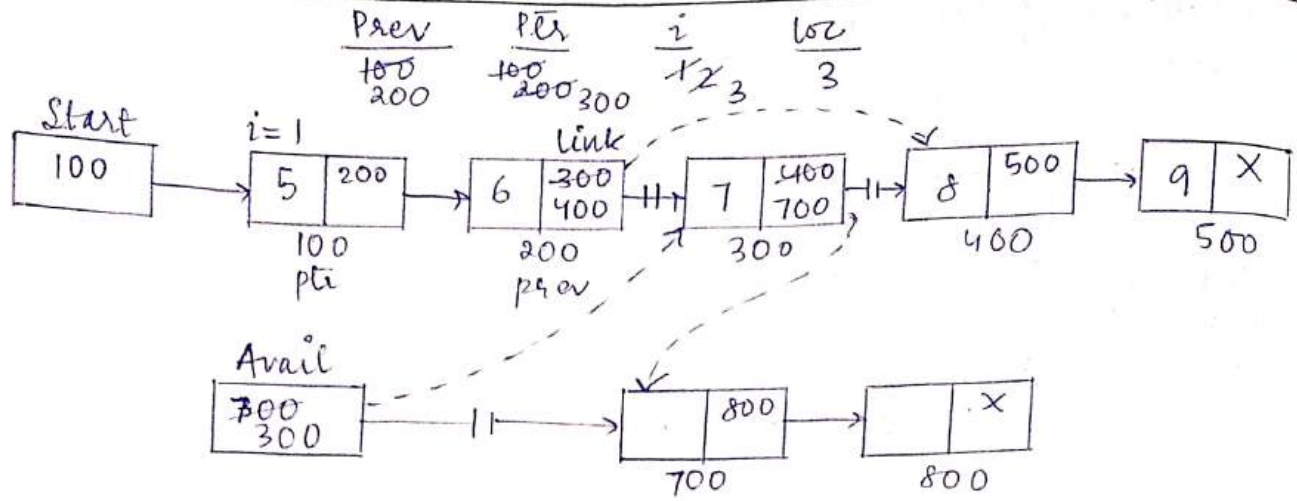
→ We can delete a node in between the linked list in 2 ways:

- Deleting a node when a node number/location is given.
- Deleting a node when a search item is given

Deletion of a node based on the node location:

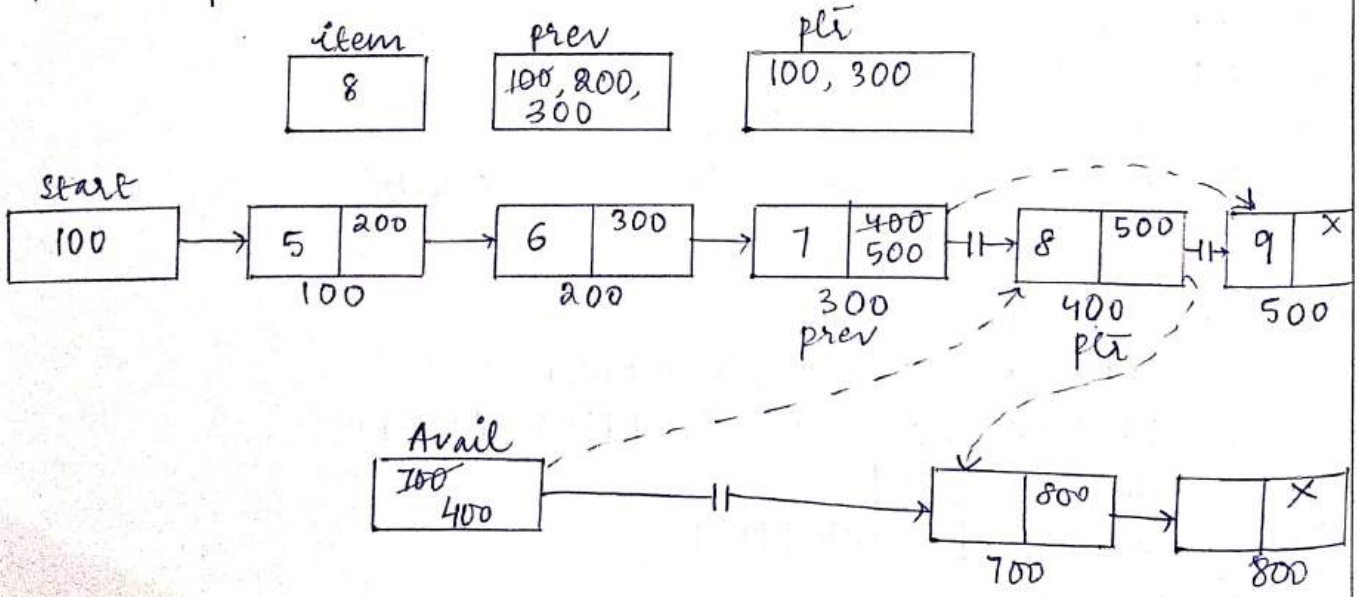
Deletion location (start, loc)

Step 1: Start
Step 2: let prev, ptr, i
Step 3: if (start = NULL), then
  Step 3.1: Display "linked list is not exist"
  Step 3.2: Exit
    [end of if]
Step 4: i = 1, ptr = start
Step 5: while (i < loc AND ptr ≠ NULL)
  Step 5.1: prev = ptr, ptr = link [ptr], i = i + 1
    [end of while]
Step 6: link [prev] = link [ptr]
Step 7: free (ptr)
Step 8: Stop



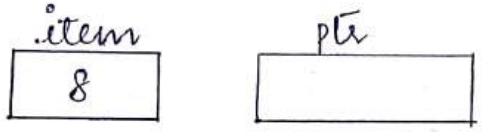
Deletion of a node based on search item:

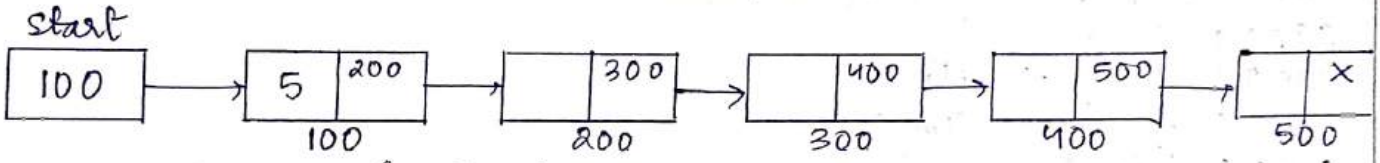
```
Deletion (Start, item)
Step 1: Start
Step 2: let, ptr, prev
Step 3: ptr = start
Step 4: while (ptr ≠ NULL AND item ≠ info[ptr])
    Step 4.1: prev = ptr
    Step 4.2: ptr = link[ptr]
        [end of while]
Step 5: if (ptr = NULL) then
    Step 5.1: Display "item not found"
Step 6: else
    Step 6.1: link[prev] = link[ptr]
    Step 6.2: free[ptr]
        [end of if]
Step 7: Stop
```



Searching for a node item:

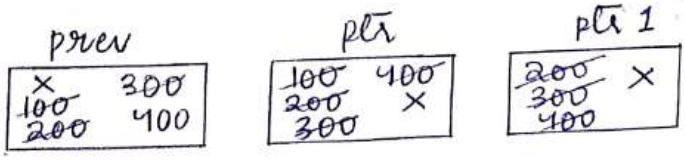
Search (start, item)
Step 1: Start
Step 2: let ptr
Step 3: ptr = start
Step 4: while (ptr ≠ NULL AND item ≠ info [ptr])
  Step 4.2: ptr = link [ptr]
    - [end of while]
Step 5: if (ptr = NULL) then
  Step 5.1: Display "item not found".
Step 6: else
  Step 6.1: Display "item found at ", ptr
    [end of if]
Step 7: Stop

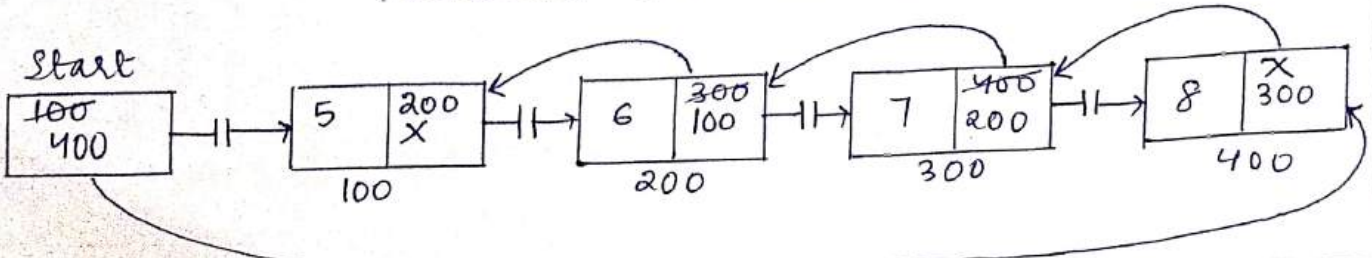




Reversing a linked list:

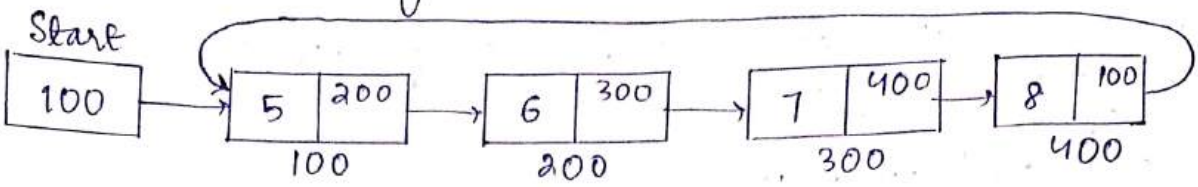
reverse (start)
Step 1: Start
Step 2: let ptr, ptr 1, prev
Step 3: prev = NULL , ptr = start
Step 4: while (ptr ≠ NULL)
  Step 4.1: ptr 1 = link [ptr]
  Step 4.2: link [ptr] = prev
  Step 4.3: prev = ptr
  Step 4.4: ptr = ptr 1
    [end of while]
Step 5: Start = prev
Step 6: Stop





Circular Single Linked list

→ A single linked list which is connected in circular manner and allows to perform operations circularly is called circular single linked list.

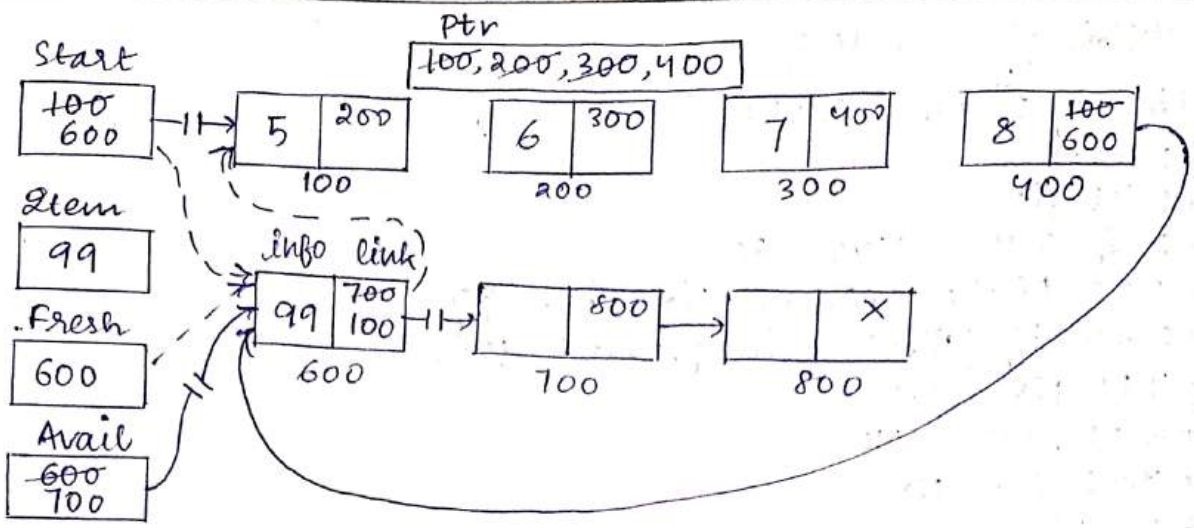


The common operations are :

1) Insertion at the beginning
2) Insertion at the end
3) Deletion from beginning
4) Deletion from end.

A. Insertion at the beginning :

```
Insertion at first (start, item)
Step 1 : start
Step 2 : let fresh, ptr
Step 3 : fresh = malloc ( )
Step 4 : info [fresh] = item
Step 5 : link [fresh] = NULL
Step 6 : if (start = NULL) then
   Step 6.1 : start = fresh
   Step 6.2 : link [start] = fresh
Step 7 : else
   Step 7.1 : ptr = start
   Step 7.2 : while (link [ptr] ≠ start)
      Step 7.2.1 : ptr = link [ptr]
            [end of while]
   Step 7.3 : link [fresh] = start
   Step 7.4 : start = fresh
   Step 7.5 : link [ptr] = start
         [end of if]
Step 8 : Stop
```

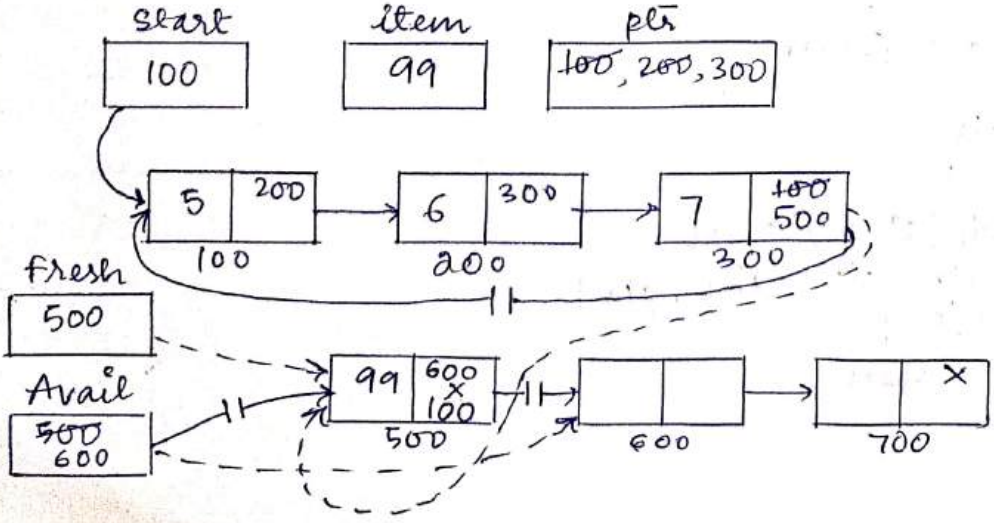


B. Insertion at the end :

Insertionatend (start, item)

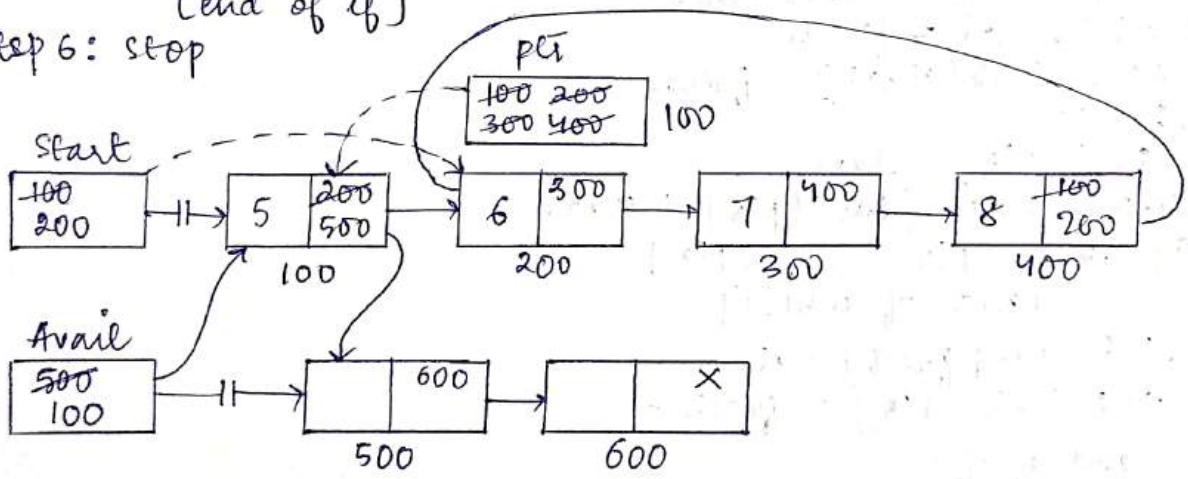
Step 1 : Start
Step 2 : let fresh, ptr
Step 3 : fresh = malloc ( )
Step 4 : info [fresh] = item
Step 5 : link [fresh] = NULL
Step 6 : if (start = NULL) then
  Step 6.1 : start = fresh
  Step 6.2 : link [start] = fresh
Step 7 : else
Step 7.1 : ptr = start
Step 7.2 : while (link [ptr] ≠ start)
  Step 7.2.1 : ptr = link [ptr]
    [end of while]
Step 7.3 : link [fresh] = start
Step 7.4 : link [ptr] = fresh
  [end of if]
Step 8 : stop

} meant for searching for the last node



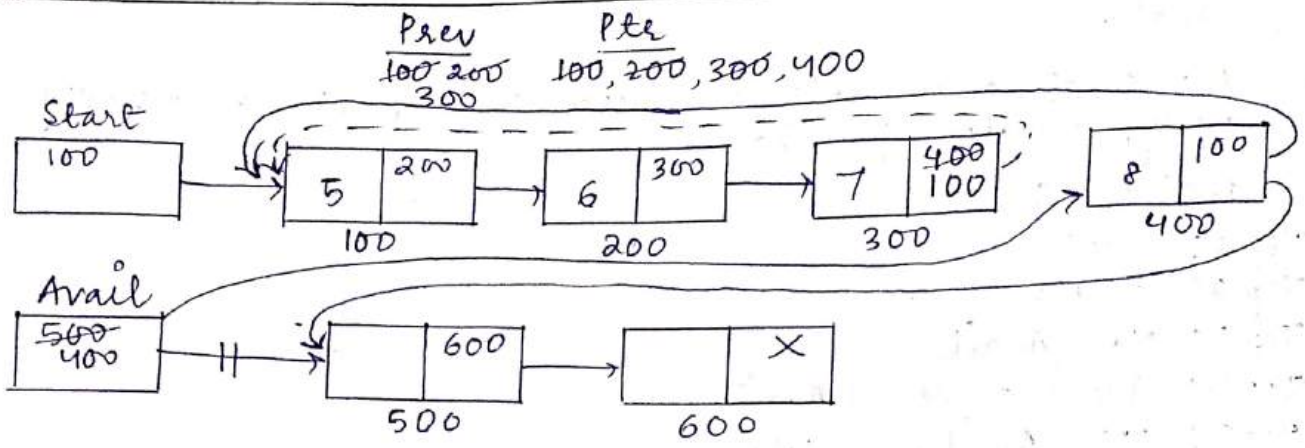
C. Deletion of 1st node :

Deletion of 1st node (start)
step 1: start
step 2: let ptr
step 3 : if (start = NULL) then
  step 3.1 : Display "linked list is empty"
step 4 : else if (start = link[start]) then
  Step 4.1 : ptr = start
  step 4.2 : start = NULL
  Step 4.3 : free (ptr)
step 5 : else
  Step 5.1 : ptr = start
  Step 5.2 : while (link[ptr] ≠ start)
    Step 5.2.1 : ptr = link[ptr]
      [end of while]
  Step 5.3 : link[ptr] = link[start]
  Step 5.4 : ptr = start
  Step 5.5 : start = link[start]
  Step 5.6 : free(ptr)
      [end of if]
step 6: stop



D. Deletion of last node :

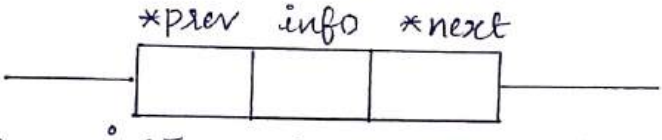
deletion of last node (start)
step 1: start
step 2: let prev, ptr
step 3 : prev = start
step 4 : while (link[ptr] ≠ start)
  step 4.1 : prev = ptr , ptr = link[ptr]
    [end of while]
step 5: link[prev] = start
step 6: free (ptr)
step 7: stop



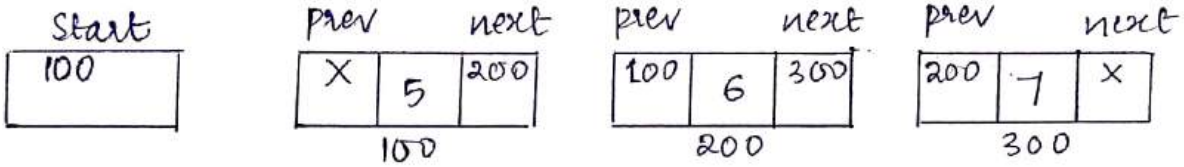
Double Linked list

→ It is a linked list in which traversing is possible at both ends.

→ Here, each node will be as:



Here *next pointer holds address of next nodes.
Similarly, *prev pointer holds address of previous node.
So, the linked list is:



→ Here, traversing is possible both in forward and backward directions. The common operations on double linked list are:

1) Creation
2) Insertion at beginning
3) Insertion at end
4) Insertion after location (or) node
5) Deletion from beginning
6) Deletion from end
7) Deletion of a specific node (or) from location
8) Searching
9) Traversing

A. Creation

→ For creating a node, we need to allocate a memory using malloc function.

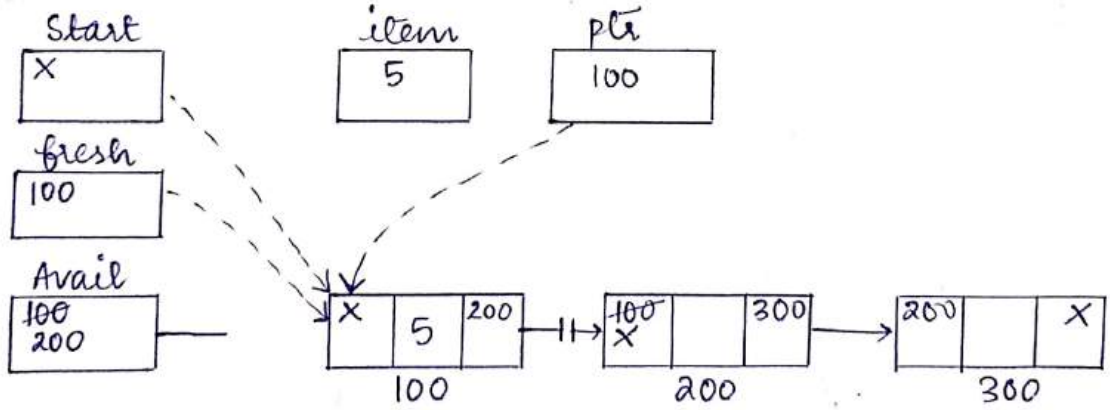
malloc (avail)

Step 1: Start
Step 2: Let ptr
Step 3: ptr = Avail
Step 4: Avail = Next [Avail]
Step 5: prev [Avail] = NULL
Step 6: return ptr
Step 7: Stop

Creation of 1st node :

Creation (start, item)

Step 1: Start
Step 2: Let fresh
Step 3: fresh = malloc()
Step 4: info [fresh] = item
Step 5: next [fresh] = NULL
Step 6: prev [fresh] = NULL
Step 7: if (start = NULL), then
  Step 7.1: start = fresh
  [end of if]
Step 8: Stop

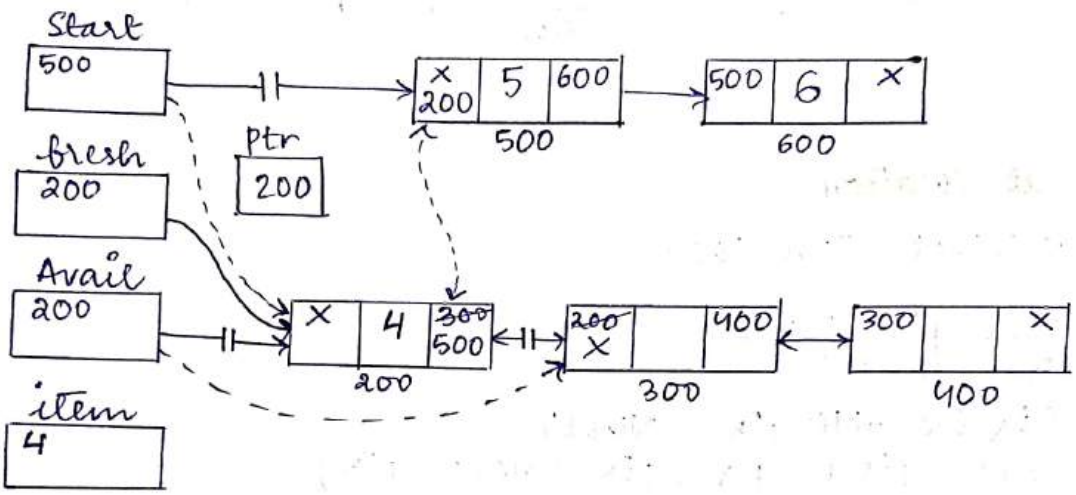


B. Insertion at the beginning

Insertfirst (start, item)

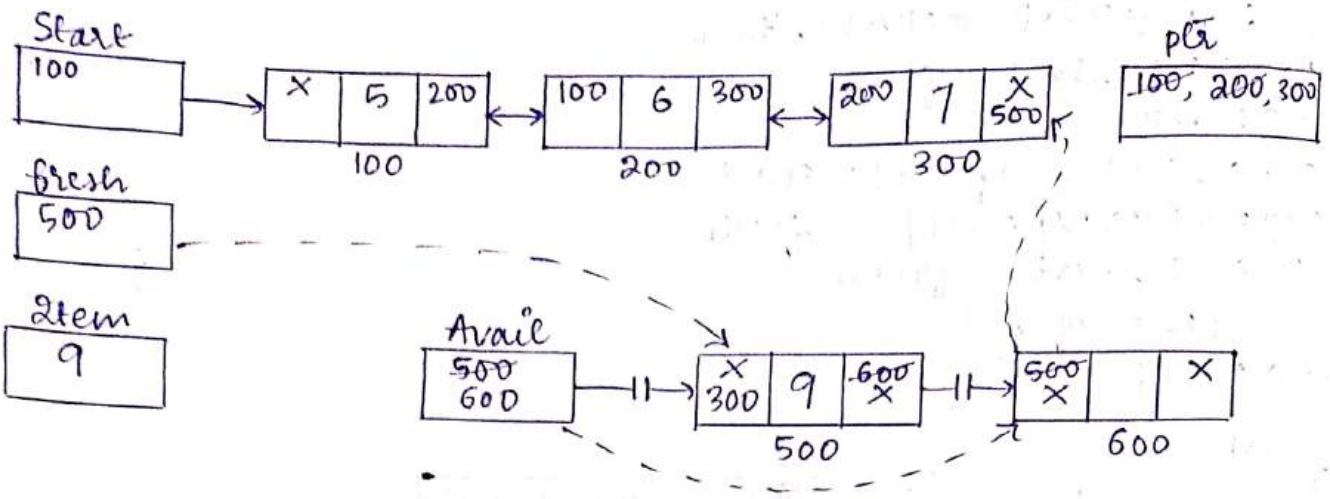
Step 1: Start
Step 2: Let fresh
Step 3: fresh = malloc()
Step 4: info [fresh] = item
Step 5: next [fresh] = NULL

Step 6 : prev [fresh] = NULL
Step 7 : if (start = NULL), then
  Step 7.1 : start = fresh
Step 8 : else
  Step 8.1 : next [fresh] = start
  Step 8.2 : prev [start] = fresh
  Step 8.3 : start = fresh
    [end of if]
Step 9 : stop



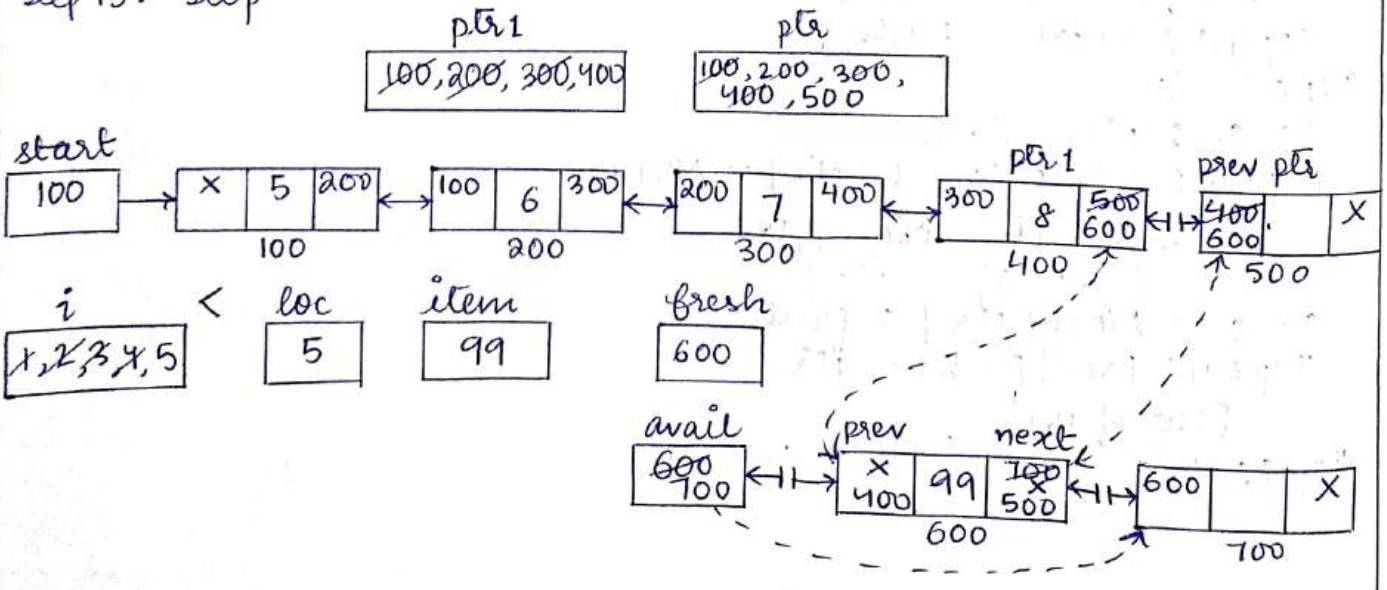
C. Insertion at the end :

Insertionatend (start, item)
Step 1 : Start
Step 2 : Let ptr, fresh
Step 3 : fresh = malloc ( )
Step 4 : info [fresh] = item
Step 5 : next [fresh] = NULL
Step 6 : prev [fresh] = NULL
Step 7 : if (start = NULL), then
  Step 7.1 : start = fresh
Step 8 : else
  Step 8.1 : ptr = start
  Step 8.2 : while (next [ptr] ≠ NULL)
    Step 8.2.1 : ptr = Next [ptr]
      [end of while]
  Step 8.3 : Next [ptr] = fresh
  Step 8.4 : Prev [fresh] = ptr
    [end of if]
Step 9 : Stop



D. Insertion at location

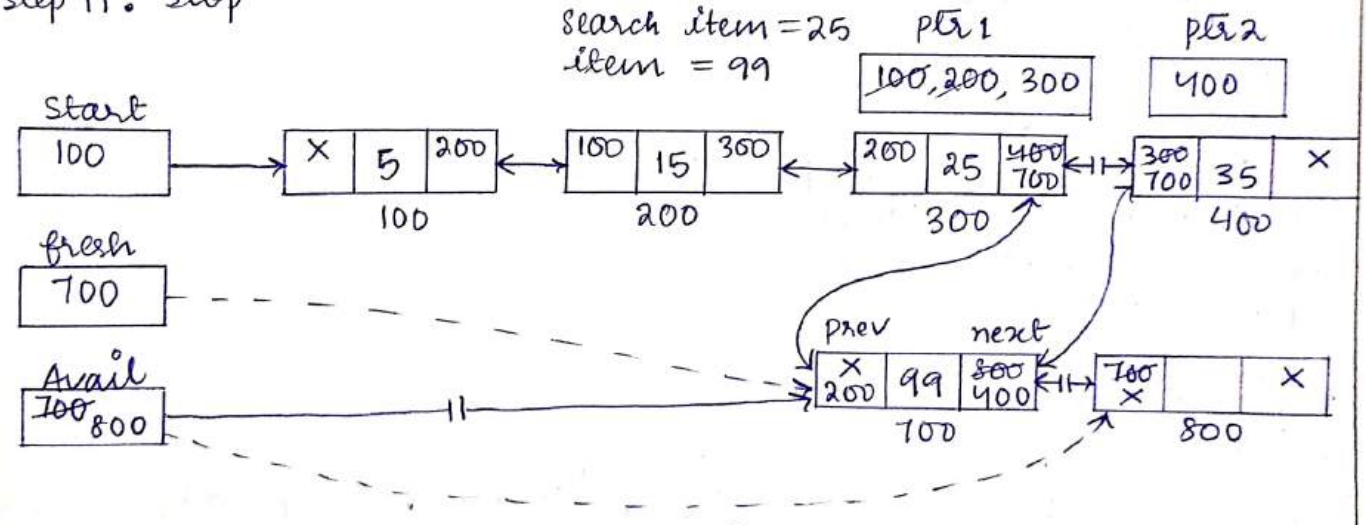
Insert at loc (start, item, loc)
Step 1: Start
Step 2: Let i, ptr1, ptr, fresh
Step 3: i=1, ptr = start
Step 4: while (i < loc AND ptr ≠ NULL)
  Step 4.1: i = i+1, ptr1 = ptr, ptr = next [ptr]
  [end of while]
Step 5: fresh = malloc ( )
Step 6: info [fresh] = item
Step 7: next [fresh] = NULL
Step 8: prev [fresh] = NULL
Step 9: next [ptr1] = fresh
Step 10: prev [fresh] = ptr1
Step 11: next [fresh] = ptr
Step 12: prev [ptr] = fresh
Step 13: stop



Insertion after search item:

insertion after search item (start, search, item, item)

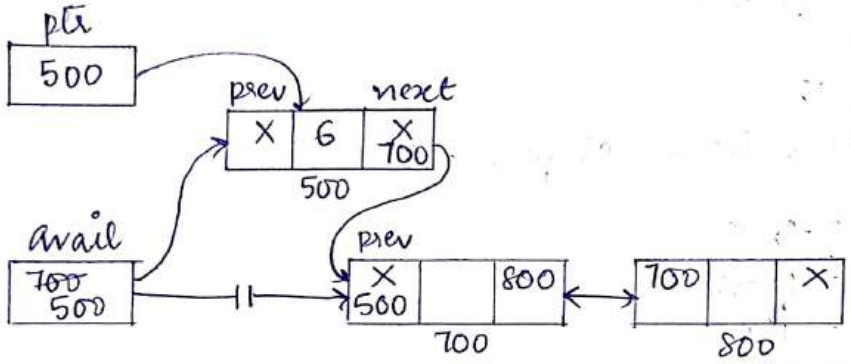
Step 1: Start
Step 2: Let ptr1, ptr2, ~~ptr~~ fresh
Step 3: fresh = malloc ( )
Step 4: info [fresh] = item
Step 5: next [fresh] = NULL
Step 6: prev [fresh] = NULL
Step 7: ptr1 = start
Step 8: while (ptr1 ≠ NULL AND info [ptr1] ≠ search item)
  Step 8.1: ptr1 = Next [ptr1]
    [end of while]
Step 9: if (ptr1 = NULL) then
  Step 9.1: display "Search item not found"
Step 10: else
  Step 10.1: ptr2 = next [ptr1]
  Step 10.2: next [ptr1] = fresh
  Step 10.3: prev [fresh] = ptr1
  Step 10.4: next [fresh] = ptr2
  Step 10.5: prev [ptr2] = fresh
    [end of if]
Step 11: Stop



Deletion

→ Deleting the node means releasing the memory and returning it to free memory area or avail list.

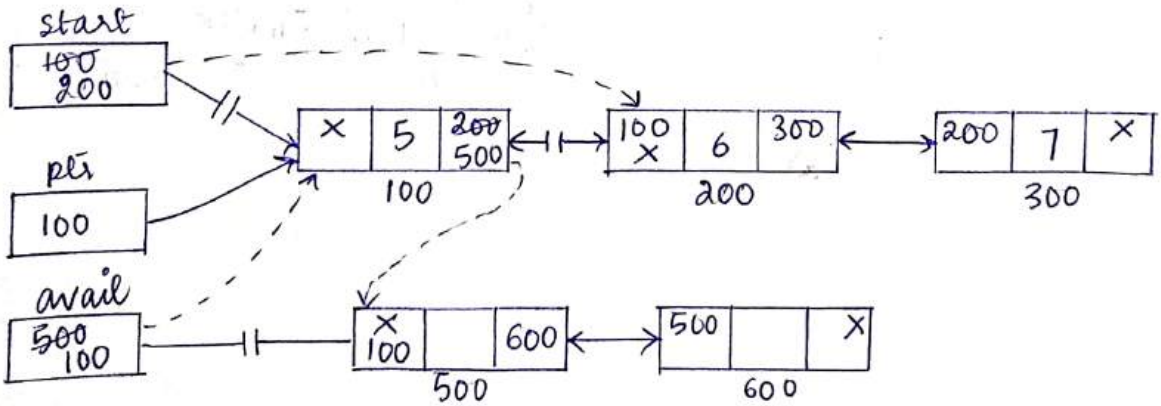
free (ptr)
Step 1: Start
Step 2: next [ptr] = avail
Step 3: prev [avail] = ptr
Step 4: Avail = ptr
Step 5: prev [avail] = NULL
Step 6: Stop



Deletion at the beginning

Deletion of 1st node :
deletion 1st (Start)

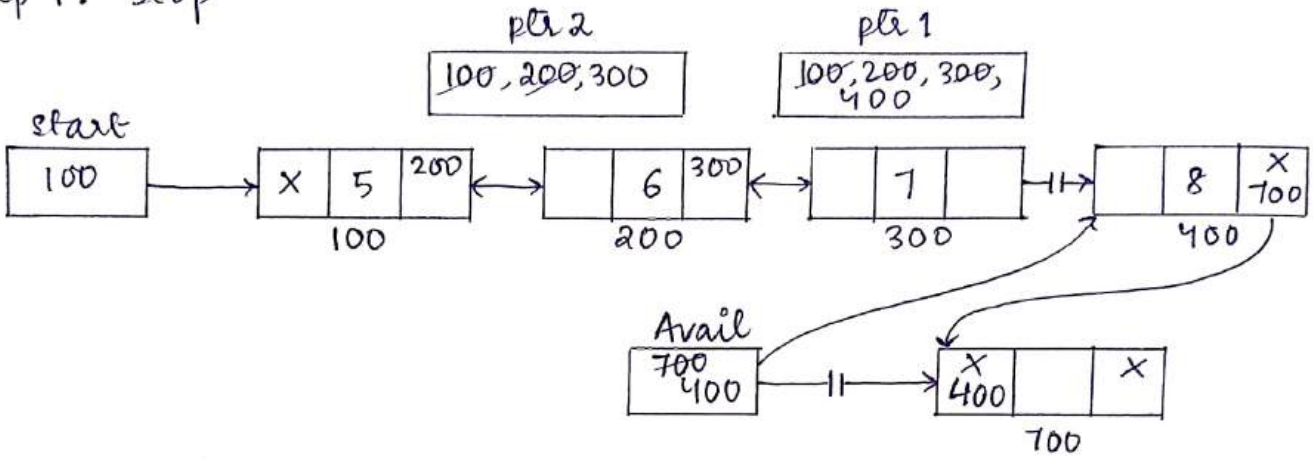
Step 1: Start
Step 2: Let ptr
Step 3: ptr = start
Step 4: start = next [start]
Step 5: prev [start] = NULL
Step 6: free (ptr)
Step 7: Stop



Deletion of last node

Deletion at last (start)

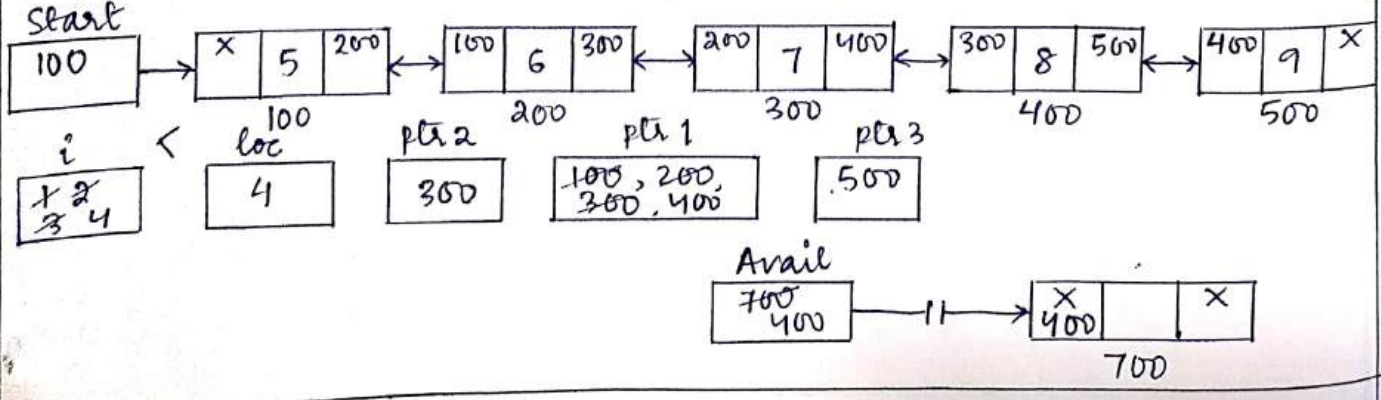
Step 1 : Start
Step 2 : Let ptr1, ptr2
Step 3 : ptr1 = start
Step 4 : while (next[ptr] ≠ NULL)
  Step 4.1 : ptr2 = ptr1, ptr1 = Next[ptr1]
  [end of while]
Step 5 : Next[ptr2] = NULL
Step 6 : free (ptr1)
Step 7 : Stop



Deletion of a node based on location :

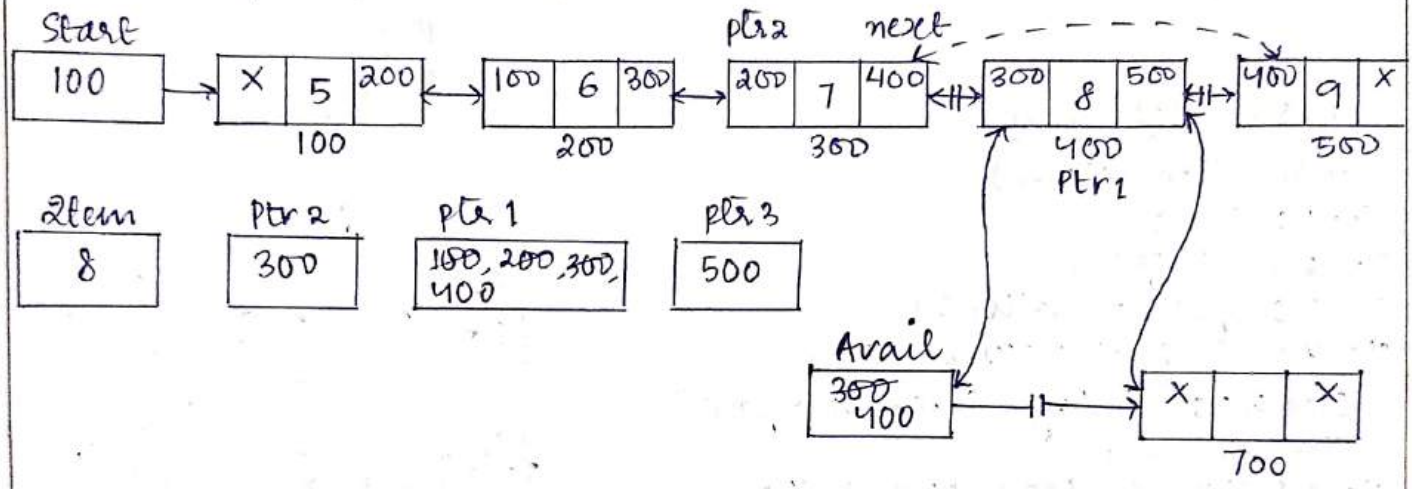
Deletion at loc (start, loc)

Step 1 : Start
Step 2 : Let ptr1, ptr2, ptr3
Step 3 : i=1, ptr1 = start
Step 4 : while (i < loc AND ptr1 ≠ NULL)
  Step 4.1 : i = i+1, ptr1 = Next[ptr1]
  [end of while]
Step 5 : ptr2 = prev[ptr1], ptr3 = Next[ptr1]
Step 6 : next[ptr2] = ptr3, prev[ptr3] = ptr2
Step 7 : free (ptr1)
Step 8 : Stop



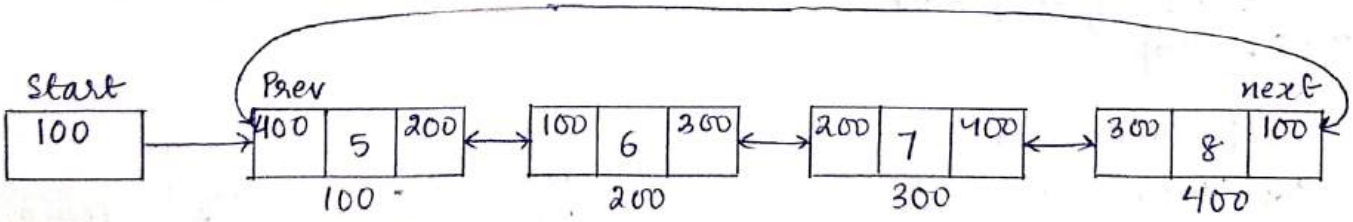
Deletion of a node based on item

Deletion of item (start, item)
step 1: start
step 2: let ptr1, ptr2, ptr3
Step 3: s ptr1 = start
step 4: while (ptr1 ≠ NULL) AND info [ptr1] ≠ item]
  Step 4.1: ptr1 = next [ptr1]
    [end of while]
step 5: ptr2 = prev[ptr1] , ptr3 = next [ptr1]
step 6: next [ptr2] = ptr3, prev [ptr3] = ptr2
step 7: free (ptr1)
step 8 : stop



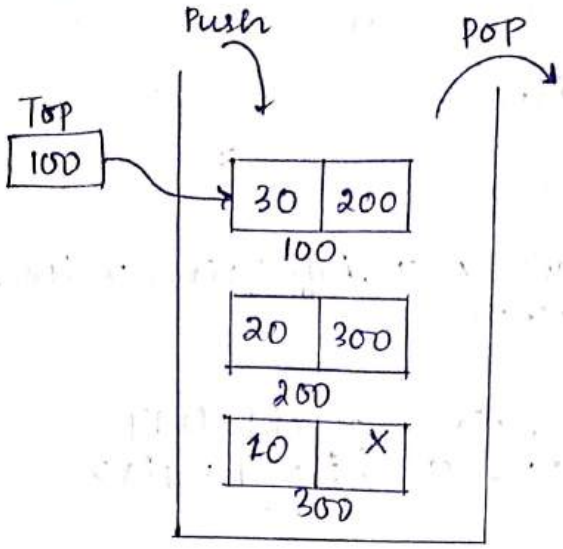
Circular Double Linked list

→ when a double linked list in connected in a circular manner, then it is known as circular double linked list.
→ In circular double linked list, traversing is possible in both forward and backward direction in circular manner.



Linked Stack

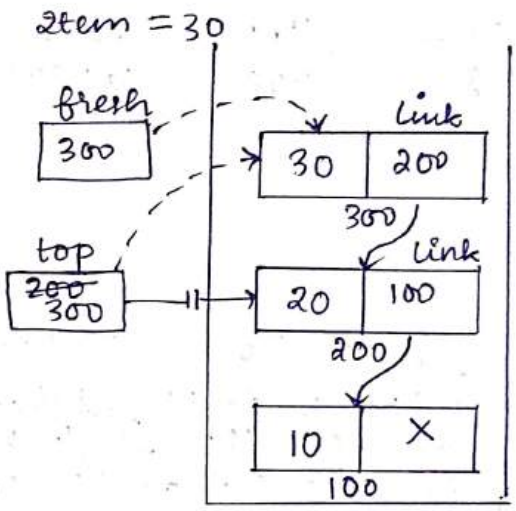
→ A single linked list which implements last in first out (LIFO) concept is called linked stack.



→ The common operations on stack are : Push
Pop
Display

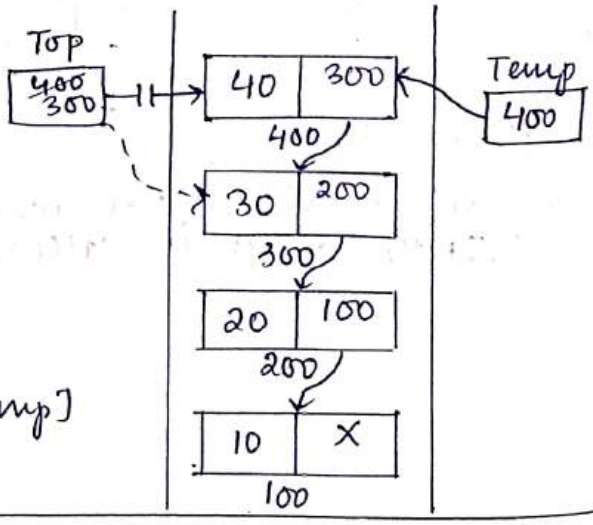
Push Algorithm :

Push (Top, item)
Step 1 : Start
Step 2 : let fresh
Step 3 : fresh = malloc()
Step 4 : if (fresh = NULL), then
  Step 4.1 : Display "Stack Overflow"
  Step 4.2 : Exit
Step 5 : else if (top = NULL), then
  Step 5.1 : top = fresh
Step 6 : else
  Step 6.1 : link [fresh] = top
  Step 6.2 : top = fresh
  [end of if]
Step 7 : info [fresh] = item
Step 8 : Stop



Pop Algorithm

Pop (top)
Step 1 : Start
Step 2 : let temp
Step 3 : if (top = NULL) then
  Step 3.1 : Display "Stack underflow"
Step 4 : else
  Step 4.1 : Temp = top
  Step 4.2 : Display "Popped", info [temp]
  Step 4.3 : top = link ['top']
  Step 4.4 : free (temp)



[end of if]
Step 5 : Stop.

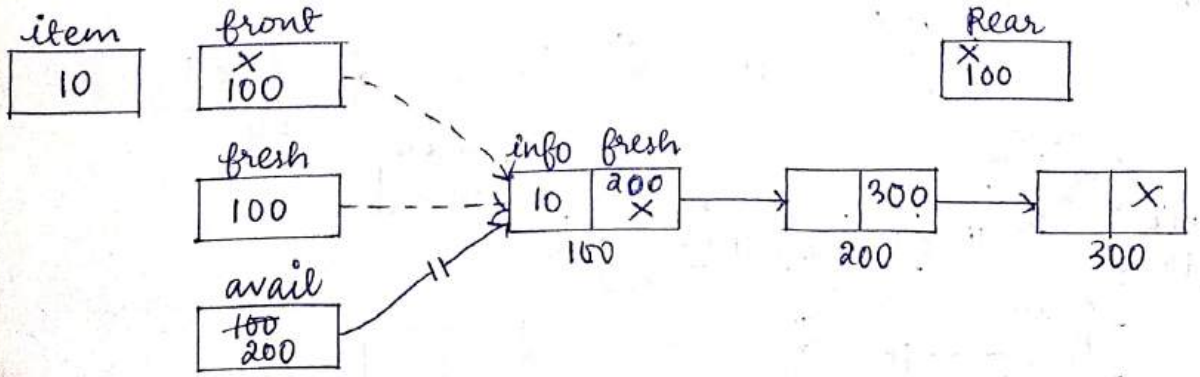
Linked Queue

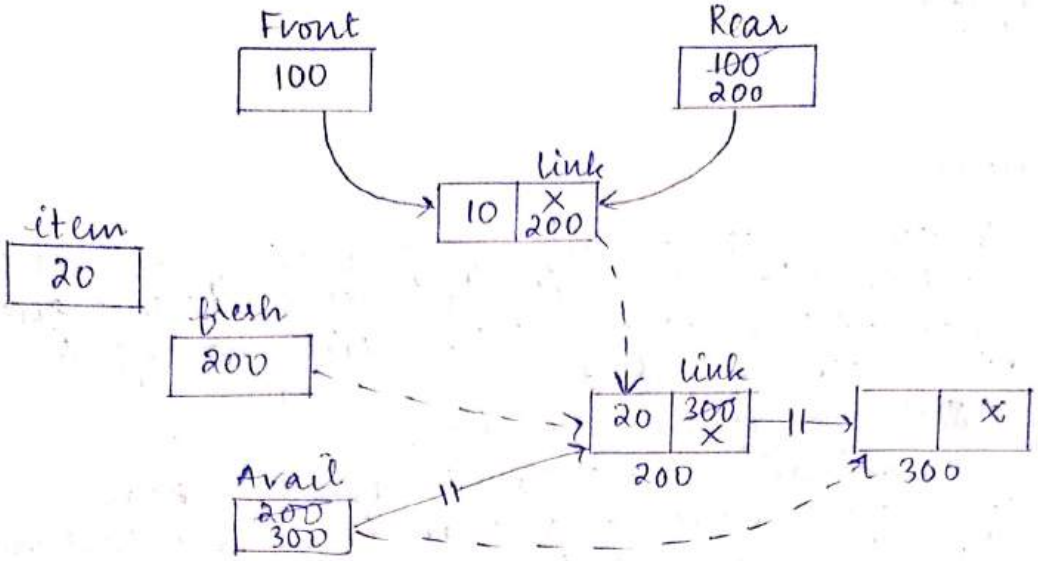
→ When we implement the concept of first in first out (FIFO) on single linked list, then it is called linked queue. So, it is equal with a single linked list having 2 operations : 1) Insertion at end 3) Display
2) Deletion from beginning.

→ Here we can maintain 2 pointers :- Front & Rear. Front identifies first node and rear identifies last node. Initially front and rear contains NULL when queue is empty.

Insertion Algorithm

Insertion (front, rear, item)
Step 1 : Start
Step 2 : Let fresh
Step 3 : fresh = malloc()
Step 4 : if(fresh = NULL), then
  Step 4.1 : Display "Queue is full
  Step 4.2 : exit
Step 5 : else
  Step 5.1 : info [fresh] = item
  Step 5.2 : link [fresh] = NULL
  Step 5.3 : if (front = NULL AND rear = NULL) then
    Step 5.3.1 : Front = fresh , rear = fresh
  Step 5.4 : else
    Step 5.4.1 : link [rear] = fresh
    Step 5.4.2 : rear = fresh
      [end of if - 5.3]
  [end of if - 4]
Step 6 : stop

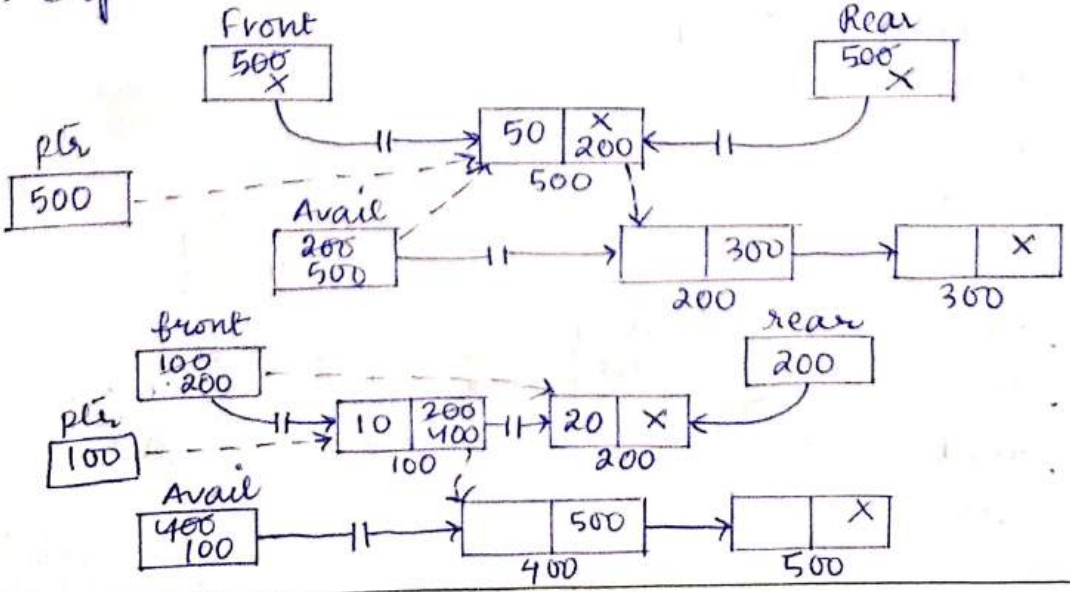




Deletion Algorithm

Deletion (front, rear)

Step 1: Start
Step 2: Let ptr
Step 3: if (front = NULL AND rear = NULL) then
  Step 3.1: Display "queue is underflow or empty"
  Step 3.2: Exit
Step 4: else if (front = rear) then
  Step 4.1: ptr = front
  Step 4.2: front = rear = NULL
  Step 4.3: free (ptr)
Step 5: else
  Step 5.1: ptr = front
  Step 5.2: front = link [front]
  Step 5.3: free (ptr)
  [end of if]
Step 6: Stop



APPLICATIONS OF LINKED LIST

→ It is used for implementing linked dictionary.

Linked Dictionary:

→ It is a dictionary of symbols or words of a language which is used to verify the syntax and semantic problems of a program.

Garbage Collection:

→ During deletion process returning back the memory node to avail list is called garbage collection. We are implementing it using free() function.

Compaction:

→ It is also called defragmentation (de-frag).

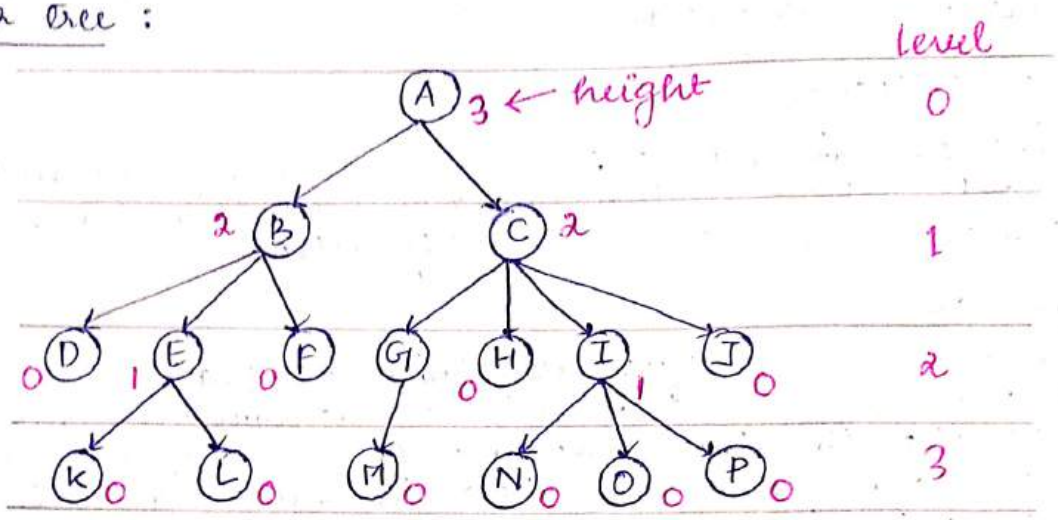
→ The process of collecting all the memory at one-end by keeping the free memory to other-end so that the execution speed can be increased is called defragmentation or compaction.

UNIT - 4 TREE

Tree :

→ It is a non-linear data structure. Here the relationship among the memory locations will be hierarchial order or top to down order.

Eg of a tree :



TREE TERMINOLOGIES :

A. **Node :**
→ Each memory location is called a node.

B. **Edge :**
→ Each connecting line is called an edge.

C. **Root node :**
→ The top most node from where all the nodes connecting is started is called root node.
Eg : In above diagram, A is the root node.

D. **Parent node :**
→ A node is called a parent node if its immediate predecessor of a node.
Eg : A is parent node of B & C.
B is parent node of D, E & F.

E. **Child node :**
→ A node is called a child node if it is immediate successor of a node.
Eg : D, E, F are child nodes of B.
B, C are child nodes of A.
K, L are child nodes of E.

F. **Sibling node :**
→ The child node belongs to a single parent are called sibling node.
Here, B and C are sibling nodes.

D, E, F are sibling nodes
N, O, P are sibling nodes

G. Leaf node :

→ The node of a tree whose child does not exist is called leaf node.

Eg: D, K, L, F, M, H, N, O, P, I are leaf nodes.

Leaf node is also called terminal node.

Rest all other nodes in tree called non-leaf nodes or non-terminal nodes.

H. Degree of a node :

→ The count of child nodes connected is called degree of a node.

Eg: Degree of node A is 2.
Degree of node B is 3.
Degree of node C is 4.
Degree of node D is 0.

Here, we can understand that all the leaf nodes are having degree 0.

I: Degree of a tree :

→ It is the highest degree available in the tree.

Eg: Here node C having highest degree 4.
So, degree of tree is 4.

J. Path :

→ A set of connecting lines from a source node to destination node is called the path.

Eg: A → C → G → M
B → E → K

K. Level :

→ Entire tree is levelled such a manner that root node is at level 0, its child nodes are at level 1, their child nodes are at level 2 and so on.

L. Depth of a node :

→ It can be easily identified based on levels.

Eg: Root node A is depth 0.
B at depth 1.

M. Height of a node :

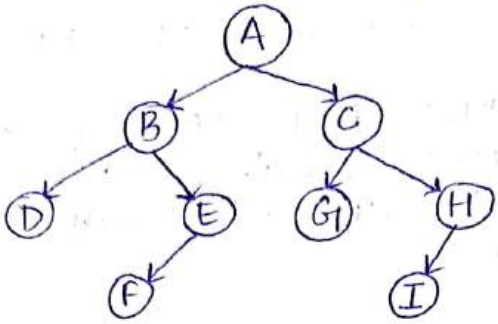
The height of each leaf node is 0.

The height of any one is count of nodes in the longest path from that node to its leaf node..

Eg: Here root node height is 3.

Binary Tree

→ A tree in which every parent node can have maximum of 2 child node is called binary tree.

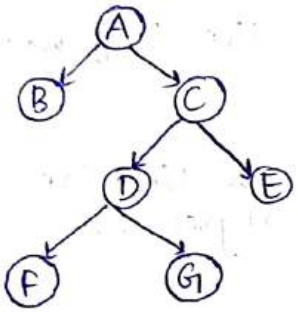


Types of Binary Tree:

1. **Strictly Binary Tree**

→ A binary tree in which every parent node can have exactly 2 child nodes is called strictly binary tree.

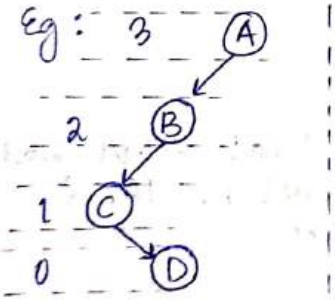
Eg:

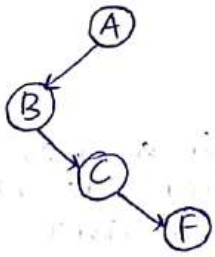


2. **Skew Binary Tree**

→ A binary tree in which every parent node can have 1 child node.

Eg:





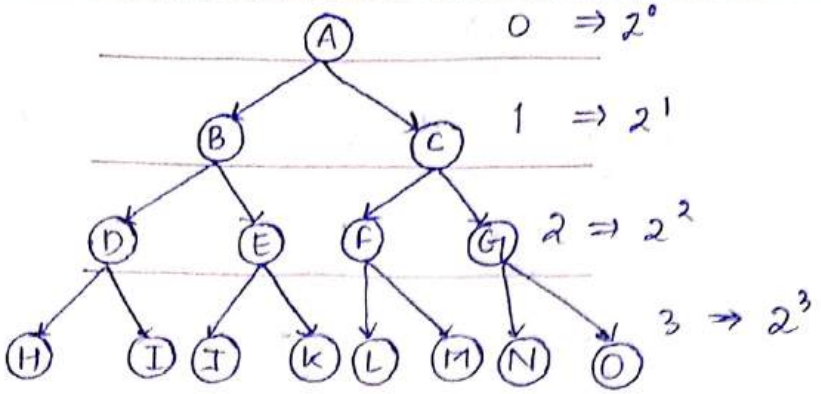
→ Here, we will find only 1 leaf node. Here, if total no. of nodes are equal to h, then height of the tree will be height − h.

3. **Full Binary Tree**

→ A binary tree is called full if it contains maximum number of nodes in all its levels.
For example, root node is at level 0, 2 child nodes at level 1, 4 nodes at level 2 and so on.

→ Maximum possible nodes at a particular level = $2^{(\text{level no})}$

Eg:

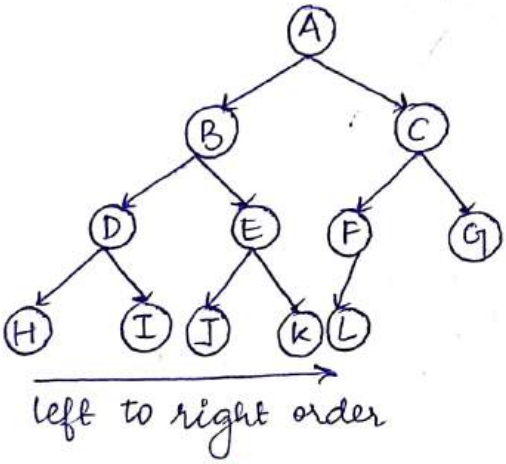


4. Complete Binary Tree

→ A binary tree is called complete if it is having maximum number of nodes in all its levels except possibly the last level.

→ In the last level, the nodes must available in left to right order.

Eg:



Memory Representation of Binary Tree

→ A binary can be stored in memory in 2 ways:

1) Array Representation OR sequential representation
2) Linked Representation

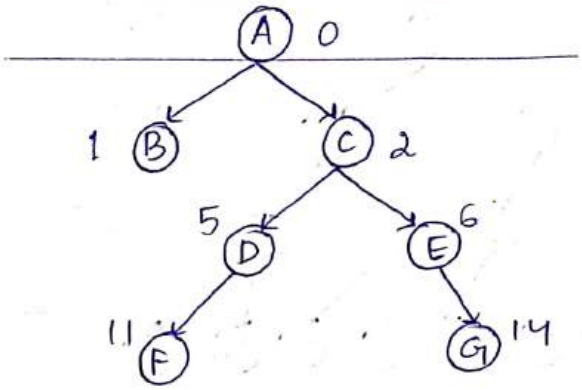
A. Array Representation

→ We can store a binary tree nodes in the following manner:

1. Store the next node at i^{th} location.
2. Store the left child at $2*i+1$ location.
3. Store the right child at $2*i+2$ location.

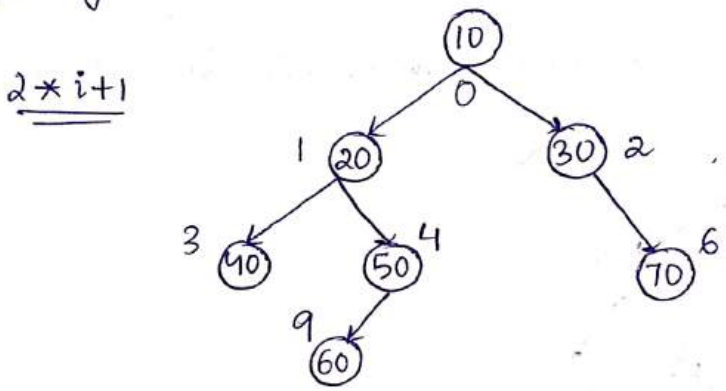
Continue this process for every parent and child nodes.

Eg:1



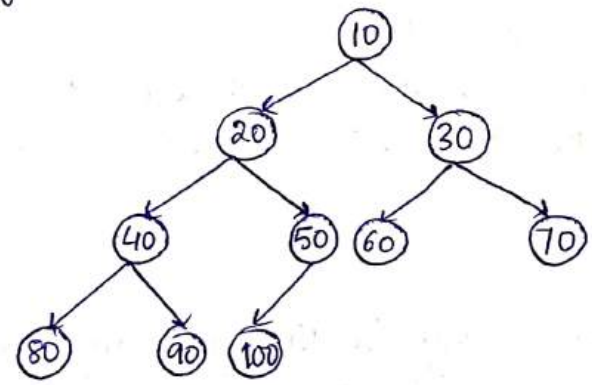
A	B	C			D	E					F			G				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	----	

Eg 2: given a tree



10	20	30	40	50		70			60			
0	1	2	3	4	5	6	7	8	9	10	11	----

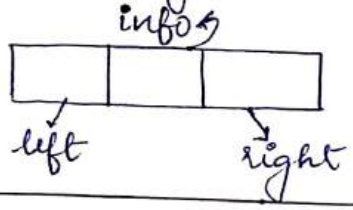
Eg 3: given a tree



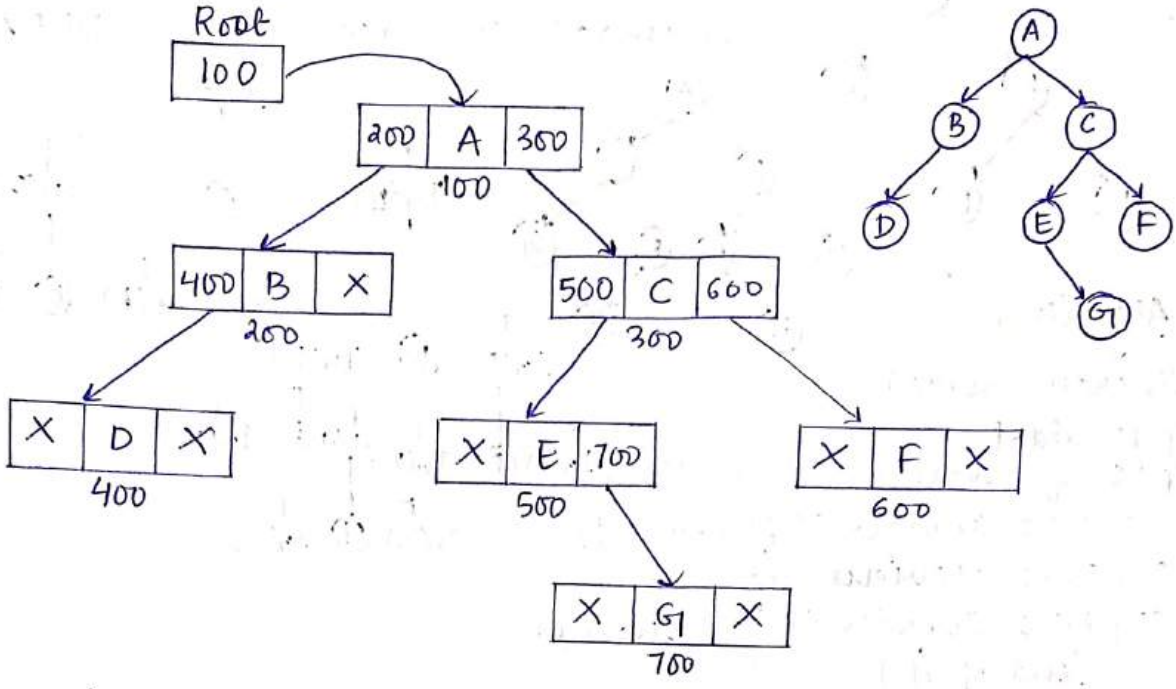
10	20	30	40	50	60	70	80	90	100			
0	1	2	3	4	5	6	7	8	9	10	11	---

Linked Representation

→ A binary tree can be stored using a collection of nodes.
A node contains
1) info
2) left pointer
3) right pointer



Consider a binary tree to represent in linked form.



Operations on Binary Tree

1) Building a tree
2) Tree traversal methods
3) Searching for a node
4) Insertion of a node
5) Deletion of a node.

Tree Traversal Methods:

They are 3 ways or types:

1) Inorder Traversal Method
2) Pre order Traversal Method
3) Post order Traversal Method

→ Traversal operations can be in 2 ways:

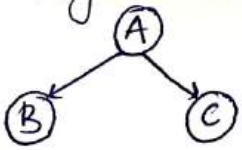
1) Recursive methods
2) Non-Recursive methods.

RECURSIVE TRAVERSAL METHODS

Inorder Traversal Method:

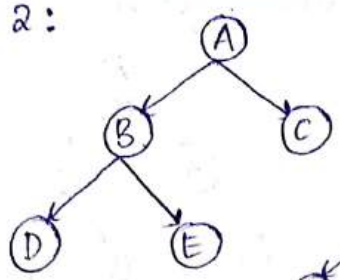
→ Here, the order of traversing the nodes will be: left, root, right order.

Eg:

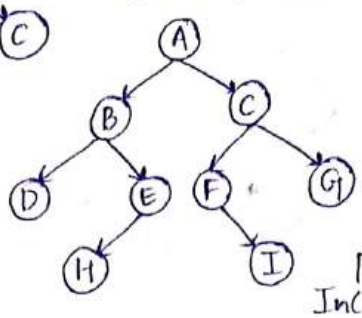


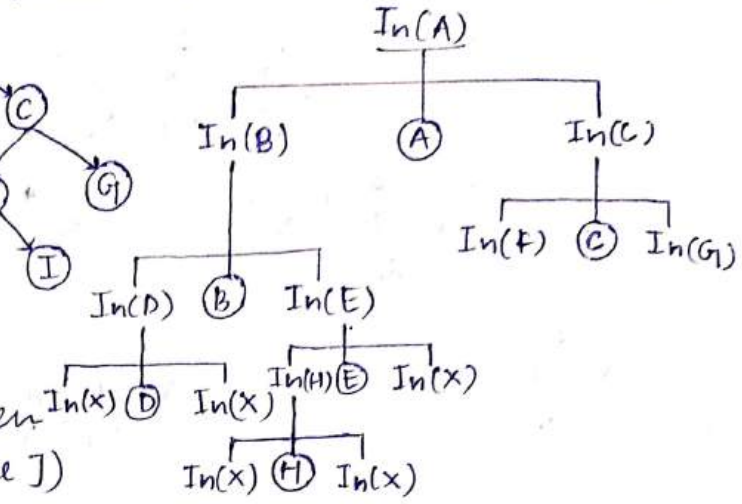
The inorder sequence will be: BAC.

Eg 2:



The inorder sequence will be : DBEAC.





Algorithm

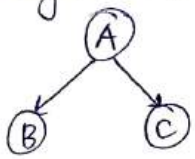
Inorder (node)

Step 1: Start
Step 2: if (node ≠ NULL) then
  Step 2.1: Inorder (left [node])
  Step 2.2: process node
  Step 2.3: Inorder (right [node])
  [end of if]
Step 3: Stop

Pre order Traversal Method:

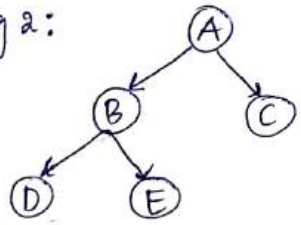
→ Here, traversal of node will be in the sequence of root, left, right order.

Eg 1:



The sequence of nodes = A B C

Eg 2:

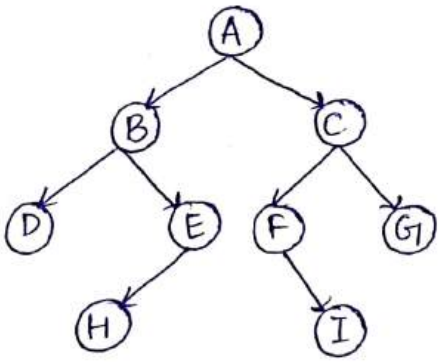


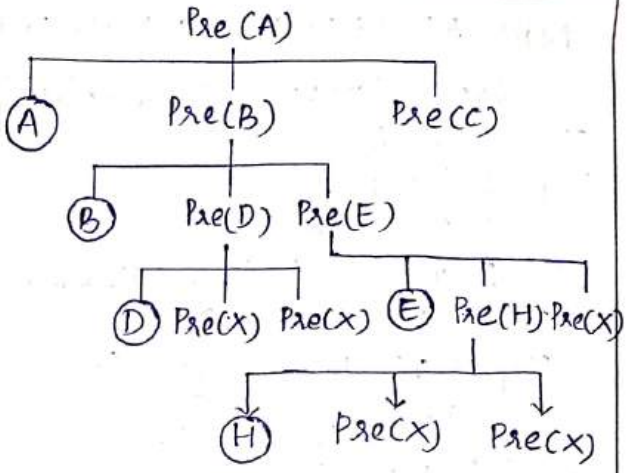
The sequence of nodes = ABDEC

Algorithm

Pre order (node)

Step 1: Start
Step 2: if (node ≠ NULL) then
  Step 2.1: process node
  Step 2.2: preorder (left [node])
  Step 2.3: preorder (right [node])
  [end of if]
Step 3: Stop



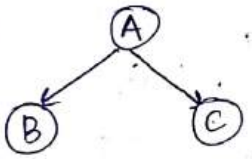


The preorder sequence of nodes : ABDEHCFIG

Postorder Traversal Method :

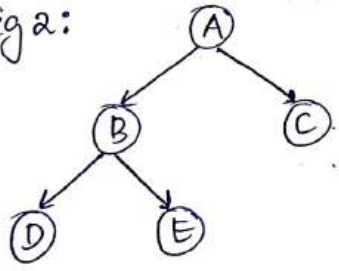
→ Here, traversal sequence of nodes will be as : left, right, root order.

Eg 1:

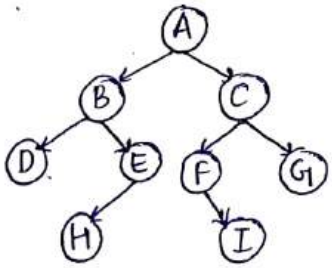


The sequence of nodes = BCA

Eg 2:



The sequence of nodes = DEBCA



Algorithm :

Postorder (node)

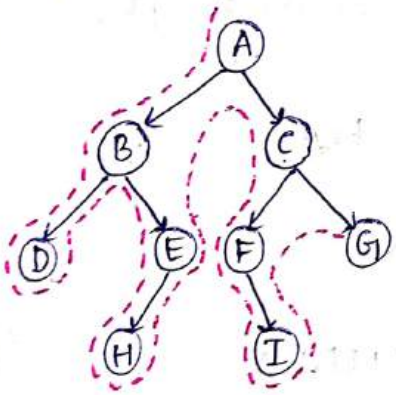
Step 1: start
Step 2: if (node ≠ NULL) then
  Step 2.1 : postorder (left [node])
  Step 2.2 : postorder (right [node])
  Step 2.3 : process node
  [end of if]
Step 3: Stop

The post-order sequence of nodes : DHEBIFGICA

NON-RECURSIVE TRAVERSAL METHODS

Inorder (root, stack [size])

Step 1: Start
Step 2: let ptr
Step 3: ptr = root
Step 4: while (ptr ≠ NULL)
  Step 4.1: push(ptr)
  Step 4.2: ptr = left [ptr]
  .. [end of while]
Step 5: ptr = pop()
Step 6: while (ptr ≠ NULL)
  Step 6.1: process ptr
  Step 6.2: if (right [ptr] ≠ NULL), then
    Step 6.2.1: ptr = right [ptr]
    Step 6.2.2: go to step 4
      [end of if]
  Step 6.3: ptr = pop()
    [end of while]
Step 7: Stop



ptr

~~A~~, ~~B~~, ~~D~~, X
~~D~~, ~~B~~, ~~H~~, ~~E~~, ~~A~~,
~~C~~, ~~F~~, X, ~~I~~, ~~C~~
~~G~~, X

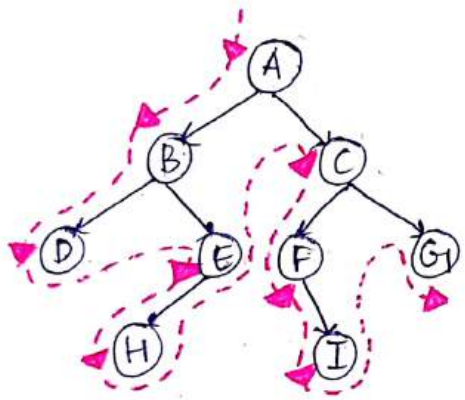
Stack

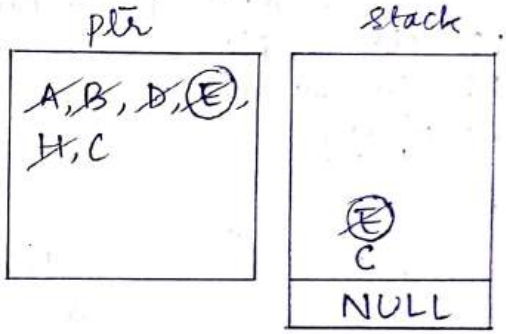
~~D~~ H
~~B~~ ~~E~~ ~~F~~ ~~I~~ ~~G~~
~~A~~ ~~C~~
NULL

Processed nodes :

→ D B H E A F I C G

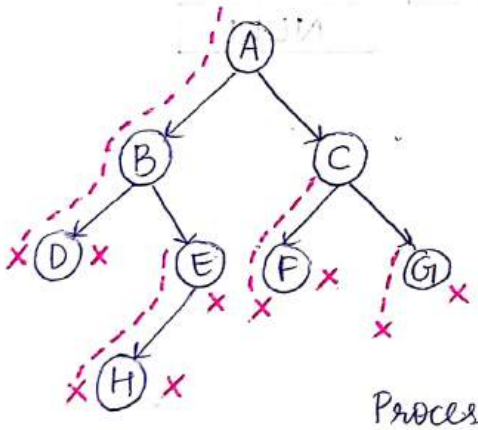
Preorder (root, stack [size])
step 1: Start
step 2: Let ptr
step 3: ptr = root
step 4: while (ptr ≠ NULL) then
  step 4.1: process ptr
  step 4.2: if (right [ptr] ≠ NULL) then
    step 4.2.1: push (right [ptr])
      [end of if]
  step 4.3: if (left [ptr] ≠ NULL) then
    step 4.3.1: ptr = left [ptr]
  step 4.4: else
    step 4.4.1: ptr = pop()
      [end of if - 4.3]
  [end of while - 4]
step 5: Stop





Processed nodes:
A B D E H C F I G

Postorder (root, stack[size])
Step 1: Start
Step 2: let ptr
Step 3: ptr = root
Step 4: while (ptr ≠ NULL) then
  Step 4.1: push (ptr)
  Step 4.2: if (right[ptr] ≠ NULL) then
    Step 4.2.1: push (-right[ptr])
      [end of if]
  Step 4.3: ptr = left[ptr]
    [end of while - 4]
Step 5: ptr = pop()
Step 6: while (ptr > 0 AND ptr ≠ NULL)
  Step 6.1: process ptr
  Step 6.2: ptr = pop()
    [end of while]
Step 7: if (ptr < 0 AND ptr ≠ NULL) then
  Step 7.1: ptr = -ptr
  Step 7.2: go to step 4
    [end of if]
Step 8: stop



ptr

~~A, B, D~~
~~D~~, ~~+E~~
~~H~~, E, B
~~+E~~, ~~F~~, ~~+G~~
G, C, A
⊗

stack

~~D~~ ~~H~~	
~~-E~~ E	~~F~~
~~B~~	~~-G~~ G
~~-C~~	
~~A~~	C
NULL	

Processed nodes → DHEBFGCA

CONSTRUCTION OF A BINARY TREE

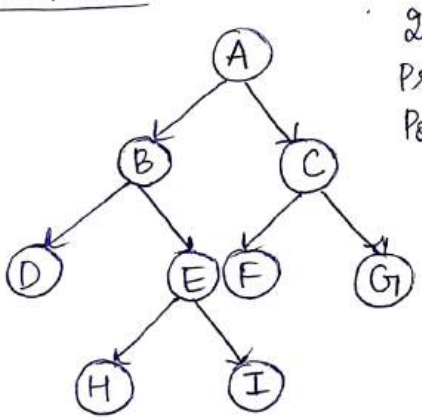
A binary tree can be constructed by using a combination of tree traversal methods such as:

1) Preorder and Inorder Traversal
2) Postorder and Inorder Traversal

Using Preorder and Inorder Traversal:

1) 1st node of the PREORDER traversal is the root node of the tree.
2) Find the position of the root node in the INORDER traversal.
   a) Nodes preceding the root in the INORDER construct the left sub tree.
   b) Nodes succeeding the root in the INORDER construct the right sub tree.
3) Find out the next roots for each sub trees using PREORDER
4) Continue the above process until leaf nodes are found.

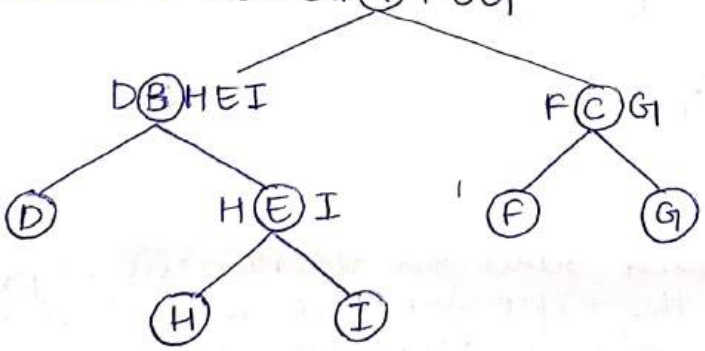
EXAMPLES:



Inorder: DBHEIAFCG (Left - root - right)
Preorder: ABDEHICFG (root - left - right)
Postorder: DHIEBFGCA (left - right - root)

Preorder: ~~ABDEHICFG~~

Inorder: DBHEI(A)FCG

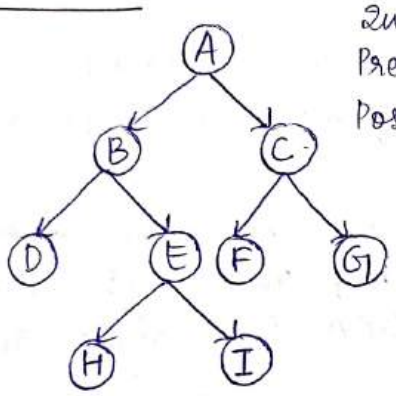


Using Postorder and Inorder traversal

PROCEDURE:

1) Here, last node in the POSTORDER traversal is considered as root node of the tree.
2) Find the position of the root node in the INORDER traversal.
   a) Nodes preceding the root in the INORDER construct the left sub tree.
   b) Nodes succeeding the root in the INORDER construct the right sub tree.
3) Find out the next roots for each sub trees using POSTORDER
4) Continue the above process until ~~left~~ leaf nodes are found.

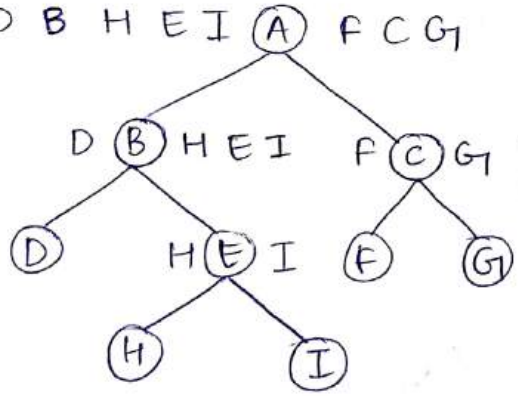
EXAMPLE:



Inorder : DBHEIAFCG (Left - root - right)
Preorder : ABDEHICFG (root - left - right)
Postorder : DHIEBFGCA (left - right - root)

Post order : ~~D H I E B F G C A~~

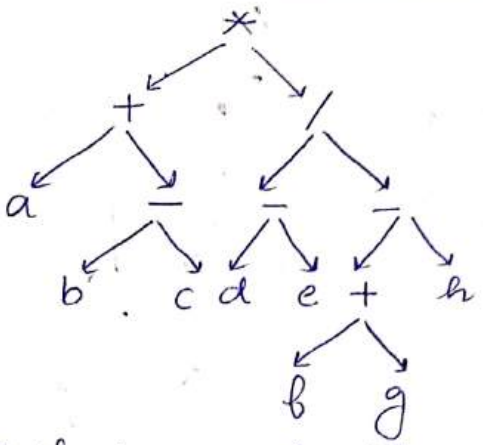
Inorder : D B H E I (A) F C G



EXPRESSION TREE:

→ It is a binary tree which stores an arithmetic expression. The leaf nodes of the expression tree are always operands and all other internal nodes are the operators including the root.

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



Divide and conquer technique used to convert an expression into a binary tree.

Construction of an Expression tree from a postfix expression

PROCEDURE:

Step 1: Scan each symbol from the postfix expression.
Step 2: If an operand is found, then create a one node tree and push a pointer to it onto the stack.
Step 3: If an operator is found, then pop two pointers belonging to two trees say T_1 and T_2 from stack and construct a tree where root is the operator, T_2 is left sub tree and T_1 is right sub tree respectively.
Step 4: Push the pointer to the above resultant sub tree onto stack.
Step 5: Exit

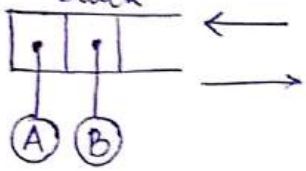
Eg: Construct and expression tree for the following postfix expression.

~~ABC*+DE^F+G/-~~

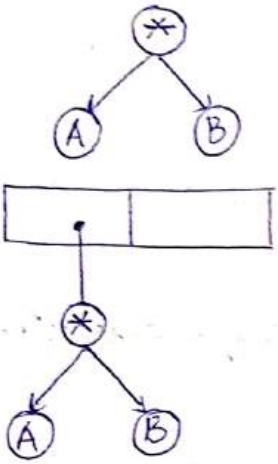
Given infix expression:

A*B-C/D^e+f
⇒ A*B-C/[De^]+f
⇒ [AB*]-C/[De^]+f
⇒ [AB*]-[CDe^/]+f
⇒ [AB*CDe^/-]+f
⇒ AB*CDe^/-f+

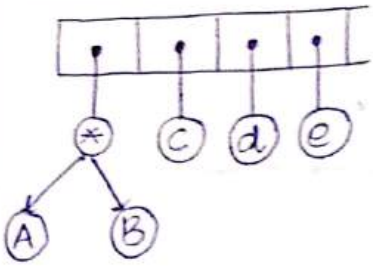
Stack



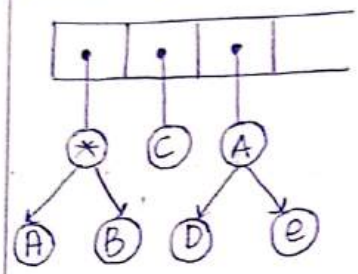
Read *:



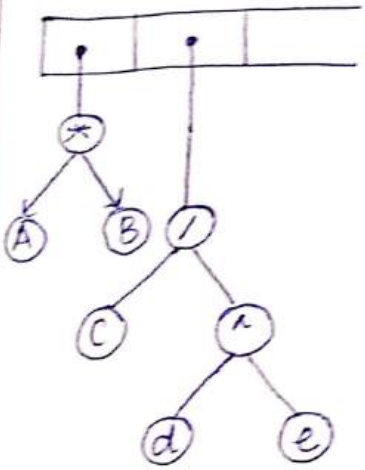
Read c, d, e:



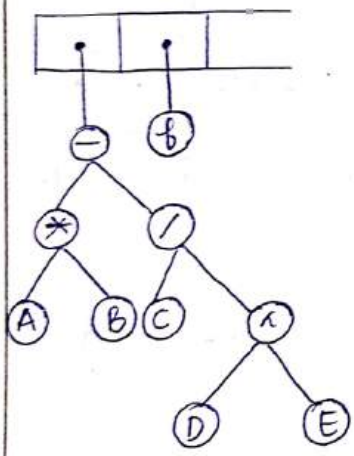
Read ^:



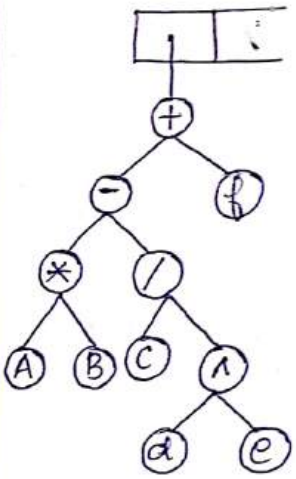
Read /:



Read -f:



Read +:



BINARY SEARCH TREE

→ A binary tree T is termed as Binary Search Tree (BST) when each left child of parent is less than its parent node and right child of parent is greater than its parent node.

BST Operations:

1. Construction of BST using insertion of nodes.
2. Insertion of a node
3. Deletion of a node
4. Searching for a node
5. Traversing (in-order, pre-order, post-order)

Construction of a BST during insertion:

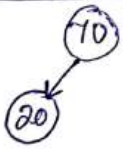
1. While constructing a BST, initially we store the first node as ROOT node, and then we follow insertion of a node based on the rules of BST.
2. During each insertion we need to search the appropriate node starting from root node.
3. If the node of insertion is less than root then go to left branch.
4. If the node of insertion is greater than root then go to right branch.

→ Binary search tree always contains a unique list of elements.

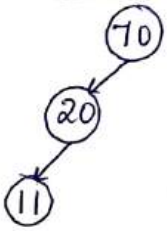
Eg: Construct a BST with the following input values:
70, 20, 11, 45, 95, 76, 29, 59.

Insert 70:

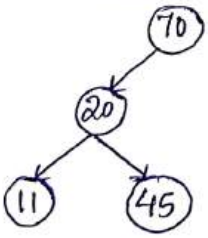
Insert 20:



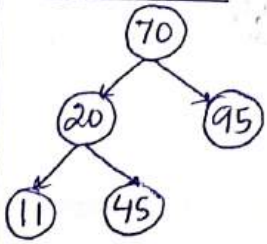
Insert 11:



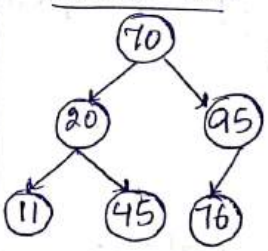
Insert 45:



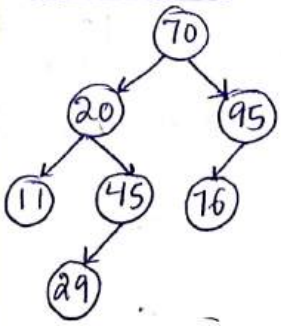
Insert 95:



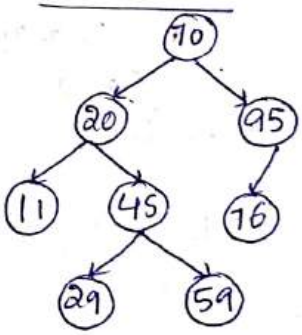
Insert 76:



Insert 29:



Insert 59:



Insertion Operation

Algorithm :

Assume root represents root node and an item given for insertion.

Insert_Bst (Root, item)

Step 1 : Start
Step 2 : Let ptr, fresh, prev
Step 3 : fresh = malloc( )
Step 4 : info [fresh] = item
Step 5 : left [fresh] = null
Step 6 : right [fresh] = null
Step 7 : if (root = NULL), then
  Step 7.1 : root = fresh
Step 8 : else
  Step 8.1 : ptr = root
  Step 8.2 : while (ptr ≠ NULL)
    Step 8.2.1 : prev = ptr
    Step 8.2.2 : if(info [ptr] > info [fresh])
                 then ptr = left [ptr]
    Step 8.2.3 : else
      Step 8.2.3.1 : ptr = right [ptr]
                [end of if]
            [end of while]
Step 9 : if (ptr = NULL), then
  Step 9.1 : if (info [prev] > info [fresh], then
    Step 9.1.1 : left [prev] = fresh
  Step 9.2 : else
    Step 9.1.2 : right [prev] = fresh
          [end of if]
        [end of if]
Step 10 : exit

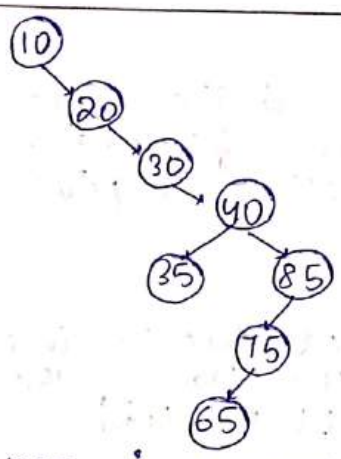
AVL Tree / Height Balanced Tree

→ In a binary search tree, the effectiveness of searching process always depends upon how data is organized to make up a specific tree.

Example of BST in skew type :

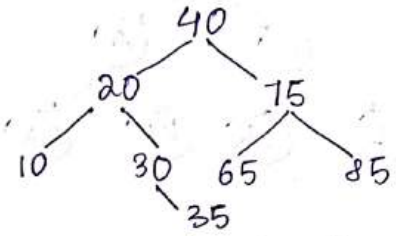
Given : 10, 20, 30, 40, 35, 85, 75, 65.

search
65



Example of BST in complete type :
given : 40, 20, 75, 10,

search
65



Here, we can understand that when a BST is skew type then the binary search works like a linear search which is not effective.
So when a BST is like a complete binary tree then only searching is effective.
So we need to keep the tree by balancing the nodes in both left and right branches to have effective searching.
We can find whether a BST is balanced or not by finding the Balance factor (B_f) of each node.

Balance Factor B_f = |Height of left branch − Height of right branch|

So, a binary search tree is height balanced if all its nodes have a balance factor of −1, 0, or +1.

→ Assume a tree is balanced initially and the tree becomes imbalanced due to insertion or deletion of a node. Now in order to make the tree balanced we need to follow certain rotation methods. The rotation methods are proposed by Russian Mathematicians G.M. Adelson, Velskii and E.M. Landis and accordingly the rotations are named as AVL rotations. So height balanced tree is also called AVL tree.

There are four types of AVL rotations. They are :

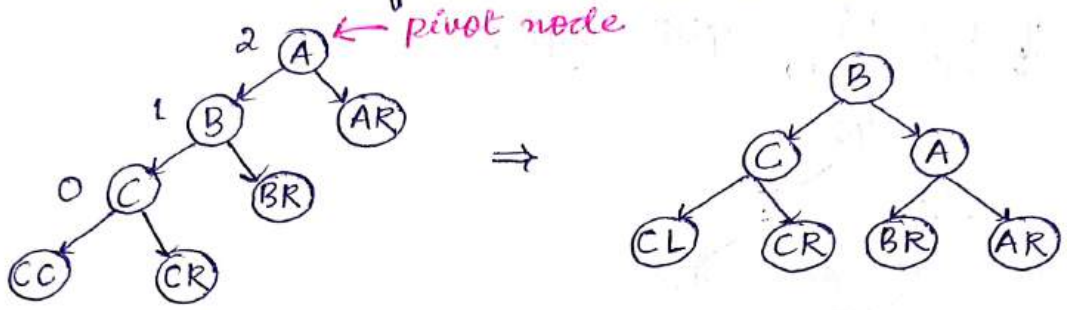
1. LL Rotation
2. RR Rotation
3. LR Rotation
4. RL Rotation

Pivot Node:

→ Pivot node of a binary tree is the node where the tree gets imbalanced i.e the node for which the balance factor $B_f = 2$ or -2 is treated as pivot node.

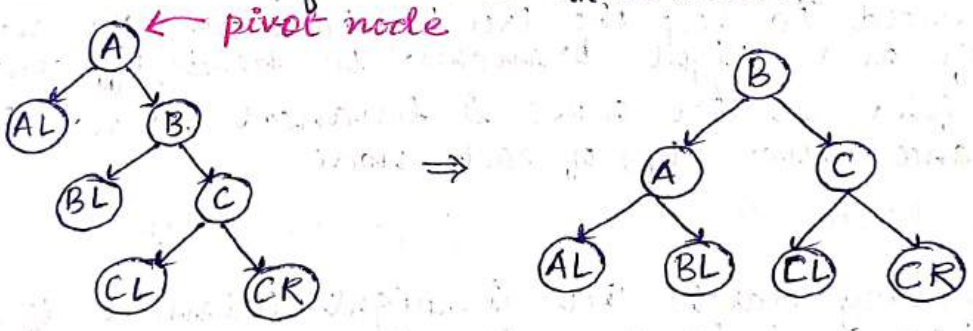
LL Rotation:

→ when the tree gets imbalanced due to insertion of a node at left side of the left sub tree of the pivot node, then we need to follow the LL rotation.



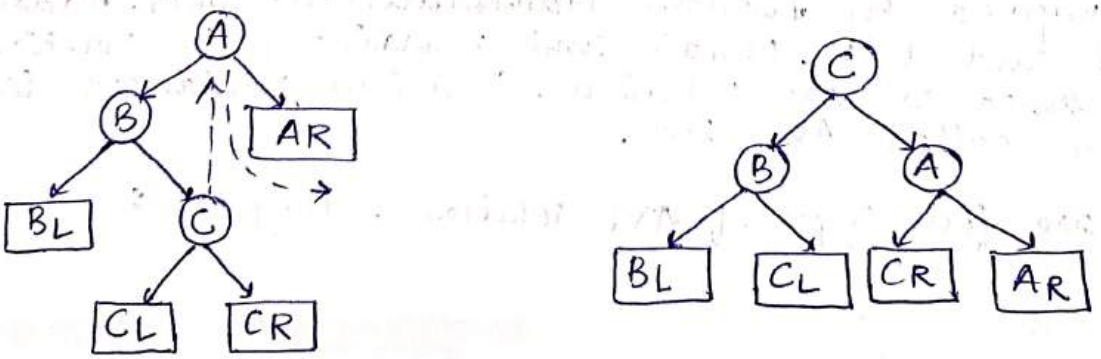
RR Rotation:

→ when the tree gets imbalanced due to insertion of a node at right side of the right subtree of the pivot node, we need to follow the RR rotation.



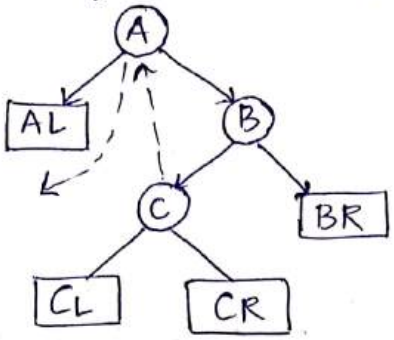
LR Rotation:

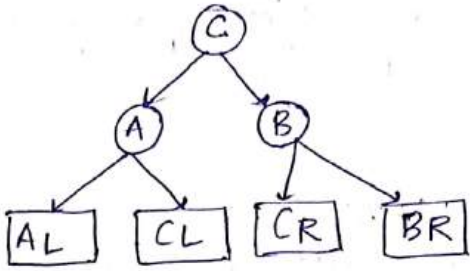
→ when tree gets imbalanced due to insertion of a node at right side of the left subtree of the pivot node, we need to follow the LR rotation.



RL rotation :

→ when tree gets imbalanced due to insertion of a node at left side of the right subtree of the pivot node, we need to follow the RL rotation.





Example on building an AVL tree

Given a list of elements for construction of an AVL tree:
10, 20, 30, 40, 80, 90, 70, 50, 25, 45

Insert 10 : 10 0 ← Pivot

Insert 20 : 10 -1, 20 0

Insert 30 : 10 -2 R, 20 -1 R, 30 0 ⇒ 20 (10, 30)

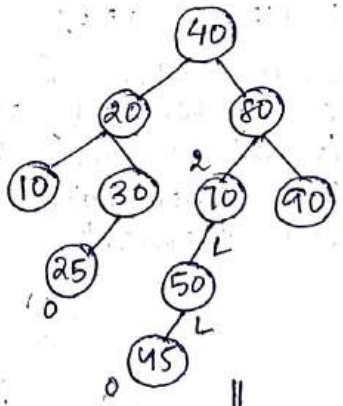
Insert 40 : 20 1-2=-1, 10, 30 -1, 40 0

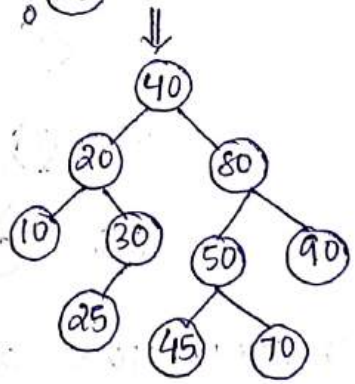
Insert 80 : 20, 10, 30 -2 R, 40 -1 R, 80 0 ⇒ 20 (10, 40 (30, 80))

Insert 90 : 20 1-3=-2 R, 10, 40 -1 R, 30, 80 -1, 90 0 ⇒ 40 (20 (10, 30), 80 (-, 90))

Insert 70 : 40 0, 20, 80 0, 10, 30, 70, 90, 70 0

Insert 50, 25, 45 :





HEAP TREE:

→ A binary tree in which from root node to leaf node if we can get an ordered list of elements then it is called a heap tree.

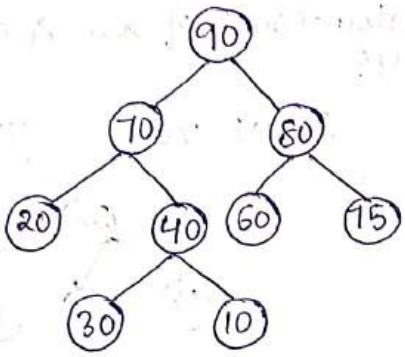
→ There are 2 types of heap tree:

1) Max Heap Tree (or) Descending Heap Tree
2) Min Heap Tree (or) Ascending Heap Tree

MAX HEAP TREE

→ A binary tree in which from root node to leaf node, if we get descending order of elements then it is called max heap tree. Here, every parent node is greater than child node.

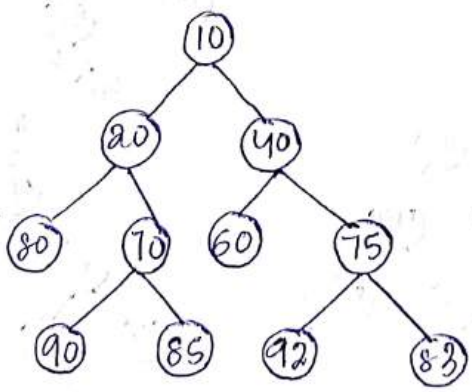
Eg:



MIN HEAP TREE

→ A binary tree in which from root node to leaf node, if we get ascending order of elements, then it is called min heap tree. Here, every parent is smaller than both child node and root node contains smallest element.

Eg:



CONSTRUCTION OF A HEAP TREE

PROCEDURE:

→ We can construct a heap tree by inserting each element in the form of a complete binary tree.

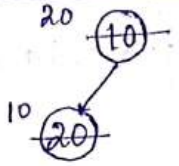
→ During insertion we can compare the element with its parent node and subsequently towards the root node for maintaining heap. For this purpose, wherever it is required, we need to apply swapping.

For the sequence of elements given construct a max heap tree —

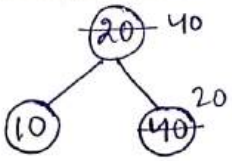
10, 20, 40, 30, 80, 60, 70, 90, 85, 45, 25, 95

Insert 10:

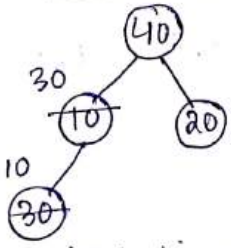
Insert 20:



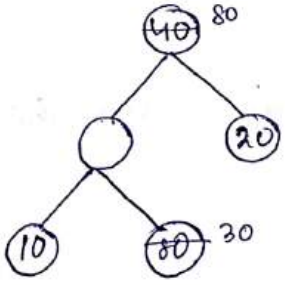
Insert 40:



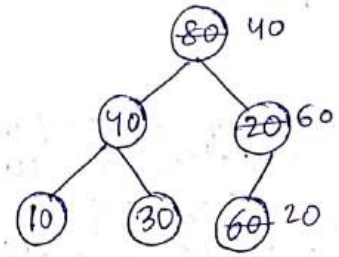
Insert 30:



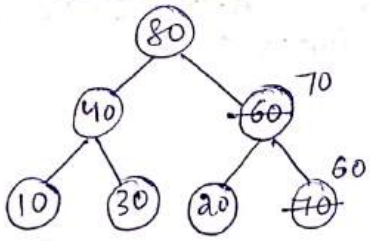
Insert 80:



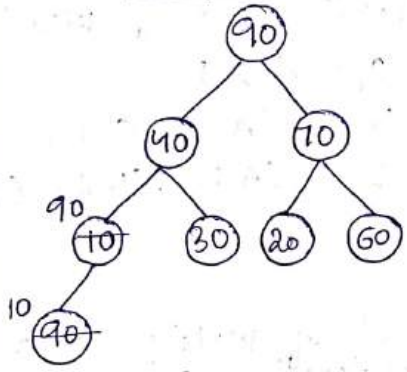
Insert 60:



Insert 70:



Insert 90:

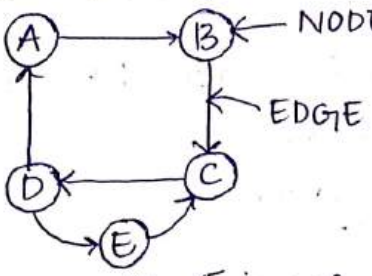


UNIT – 5 GRAPH

Graph :

→ A collection of nodes connected randomly and having random relationship among them is called graph.

Eg :



→ Graph is a collection of nodes / vertices and edges.

→ A graph can be represented using set of representation as:

Graph $G = \{V, E\}$

where $V \rightarrow$ set of vertices

$E \rightarrow$ set of edges

$E = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A, D \rightarrow E, E \rightarrow C\}$

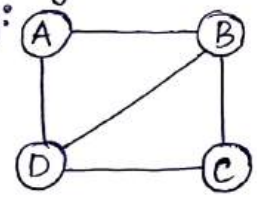
→ A graph is of 3 types :

a) Undirected graph
b) Directed graph
c) Weighted graph

A. Undirected Graph

→ A graph is a set of vertices and edges in which the edges are unordered pair of vertices.

→ Eg:



$G = \{V, E\}$

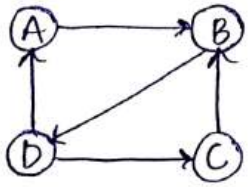
$V = \{A, B, C, D\}$

$E = \{(A,B), (B,C), (C,D), (D,A), (D,B)\}$

B. Directed Graph

→ A graph is a set of vertices and edges in which the edges are in ordered pair.

→ Eg :



$G = \{V, E\}$

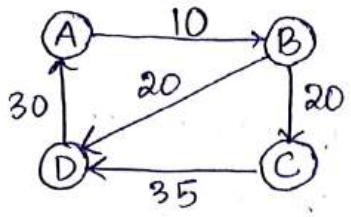
$V = \{A, B, C, D\}$

$E = \{(A,B), (C,B), (D,C), (D,A), (B,D)\}$

C. Weighted Graph

→ A graph in which every edge is having some weight, distance or cost.

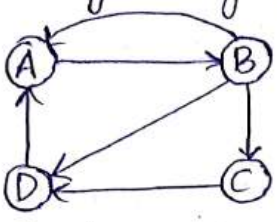
Eg:



CYCLIC GRAPH

→ A graph in which, if cycle is present, then it is called cyclic graph.

→ Eg:

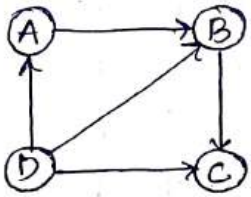


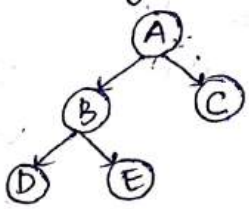
1) A → B → A
2) A → B → D → A
3) A → B → C → D → A

ACYCLIC GRAPH

→ A graph which do not have cycle is called acyclic graph.

→ Eg:





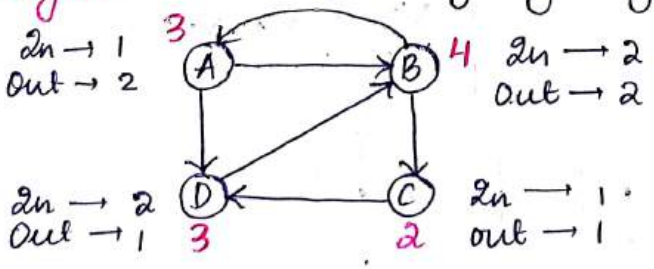
→ Every tree is an acyclic graph.

DEGREE OF A VERTEX

→ The number of edges connected to a vertex is called its degree.

→ It is of 2 types:

a) Indegree → No. of incoming edges
b) Outdegree → No. of outgoing edges



MEMORY REPRESENTATION OF GRAPH:

It can be done in 3 ways:

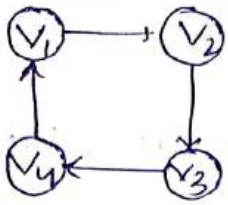
1) Matrix Representation
   1.1) Adjacency matrix
   1.2) Incidence matrix
2) Linked Representation.

Matrix Representation

1.1) Adjacency Matrix

→ When a graph G having n no. of nodes, then create a matrix with N × N size.

→ Eg:



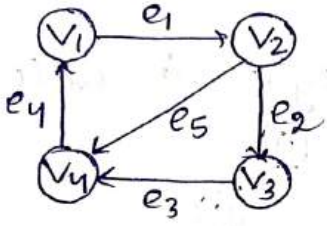
No. of nodes n = 4
Matrix A[4][4]

	V_1	V_2	V_3	V_4
V_1	0	1	0	0
V_2	0	0	1	1
V_3	0	0	0	1
V_4	1	0	0	0

1.2) Incidence matrix

→ When a graph having N no. of nodes and E no. of edges, then create a matrix with size N × E.

→ Eg:

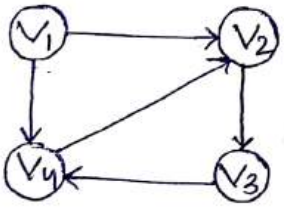


n = 4, e = 5

	e_1	e_2	e_3	e_4	e_5
V_1	1	0	0	-1	0
V_2	-1	0	0	0	1
V_3	0	-1	1	0	0
V_4	0	0	-1	1	-1

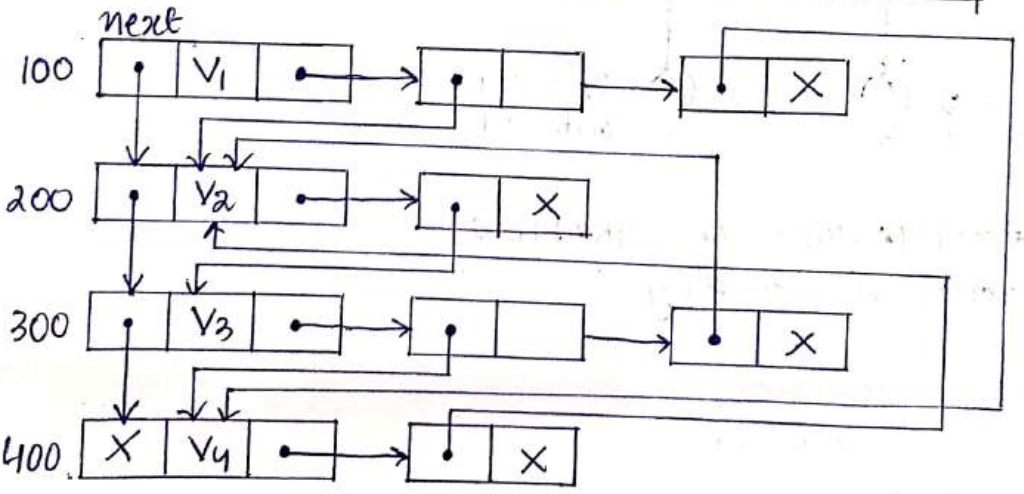
Linked Representation

→ A graph can be represented in linked form. Here we need to create a linked list for all the nodes and edges list for each node.



Adjacency Table

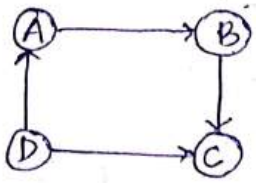
$V_1 \rightarrow V_2, V_4$
$V_2 \rightarrow V_3$
$V_3 \rightarrow V_4, V_2$
$V_4 \rightarrow V_2$



Path Matrix

→ A matrix which contain information about there exist a path or not between every pair of vertices is called path matrix.

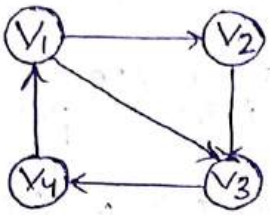
Eg1:



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

P[4][4]

Eg 2:



	V_1	V_2	V_3	V_4
V_1	1	1	1	1
V_2	1	1	1	1
V_3	1	1	1	1
V_4	1	1	1	1

P[4][4]

Here, the adjacency matrix A is:

	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	1	0
V_3	0	0	0	1
V_4	1	0	0	0

which represent path using single edge

$A \Rightarrow$ path using one edge

we can find $A^2 \Rightarrow$ path using 2 edges

$A^3 \Rightarrow$ path using 3 edges

$A^4 \Rightarrow$ path using 4 edges

Then, we can find $B = A + A^2 + A^3 + A^4$ which shows all positive roots.

Then, using matrix B, we can obtain path matrix P.

Graph Traversal Methods:

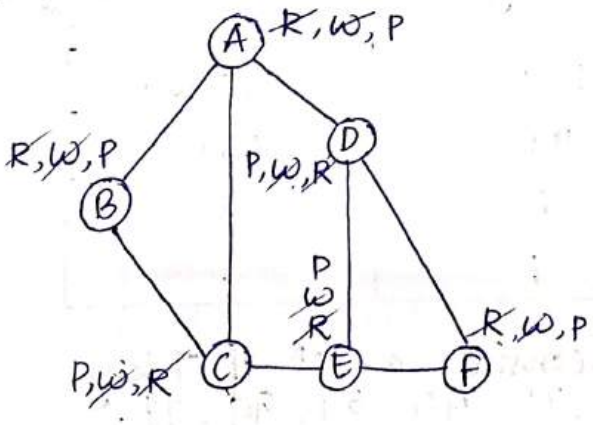
a) Depth first search Traversal (DFST)

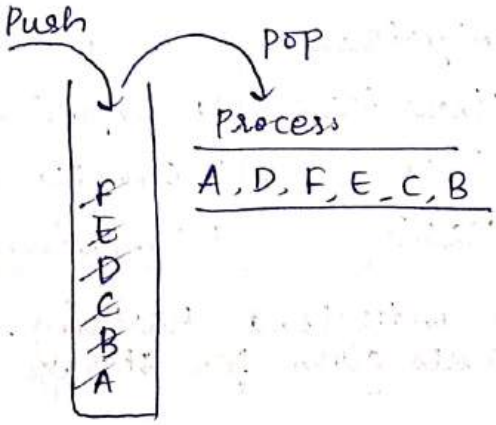
b) Breadth first search traversal (BFST)

A) Depth First Search Traversal (DFST)

To traverse all the nodes of a graph, it must be a connected one.

Here, we use stack, to traverse all the nodes.





Here, each nodes will have 3 states:

a) Ready State (R)
b) Waiting State (W)
c) Processed State (P)

Algorithm

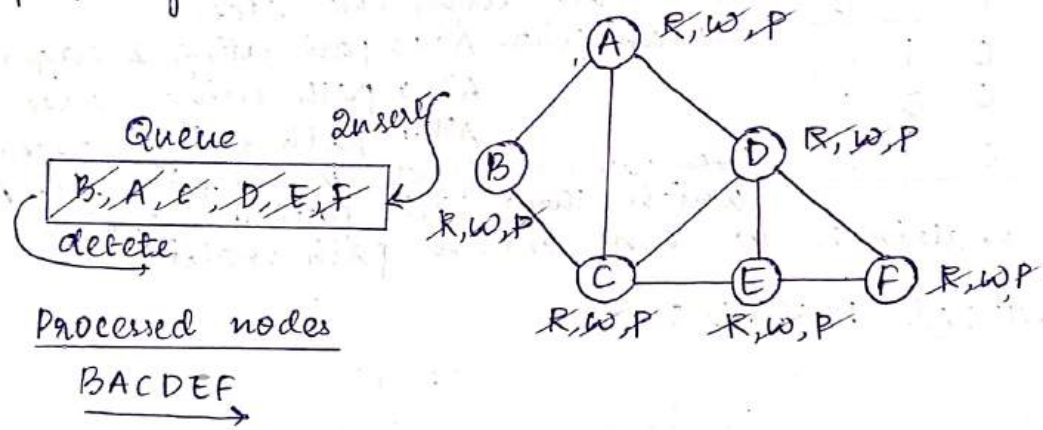
Step 1: Start
Step 2: Initialize all the nodes to ready state
Step 3: Consider a node as 1st node and push into stack.
Step 4: Change state of 1st node into waiting state.
Step 5: while (stack is not empty)
  Step 5.1: pop a node (X) and process it.
  Step 5.2: change state of node (X) into processed state.
  Step 5.3: for each adjacent node (X) of node (X) which are in ready state, push into stack and change their state to waiting state
  [end of while]
Step 6: Stop

Breadth First Search Traversal (BFS)

→ In this approach we can traverse all the nodes of a graph using a queue. Here, a graph need to be connected one.

Algorithm

Step 1 : Start
Step 2 : Initialize all the nodes to ready state.
Step 3 : Consider a node X as 1st node and insert into queue.
Step 4 : Change the state of node X into waiting state.
Step 5 : while (queue is not empty)
  Step 5.1 : delete a node X from queue and process it.
  Step 5.2 : change the state of node X into a processed state.
  Step 5.3 : for each adjacent node Y of node X which is in ready state ; insert into queue and change its state to waiting state.
  [end of while]
Step 6 : stop



Processed nodes

BACDEF →

HASHING :

→ It is a searching technique whose time complexity is O(1). Here to search for any value the time taken is constant. In the linear search the worst case time complexity is O(n). In the binary search it is $O(\log_2 n)$.

Algorithm	N = 10	N = 100	N = 1000
Linear Search	10 comparisons	100	1000
Binary Search	4 comparisons	7	8
Hashing	1 comparison	1	1

To understand hashing, let us consider a set of 7 key values given for storage : 10, 21, 34, 45, 67, 99, 77.

Now let us apply a mathematical function to generate an address :

Function is :

1) Add individual digits
2) Find unit position digits

Key	Function	Address.
10	1+0 = 1	1
21	2+1 = 3	3
34	3+4 = 7	7
45	4+5 = 9	9
67	6+7 = 13	3
99	9+9 = 18	8
77	7+7 = 14	4

Now let us consider we have a physical memory locations whose address start from 0 to 9.
Then, we can store these key values in the respective generated address.

Physical Memory

Addresses	key
0	
1	10
2	
3	(21, 67) → Collision
4	77
5	
6	
7	34
8	99
9	45

10 ⇒ 1+0 = 1

45 ⇒ 4+5 = 9

So, the hash function H : K → L.
where, K is the key and L is the hash address.

→ The standard hash functions are :

1) Division Method
2) Folding Method
3) Mid square Method

Division Method

→ Here, we need to consider a prime number M that is greater than total number of key values.
Then we need to apply function:
K%M to generate remainder as hash address.

Consider an example:
Say there are 20 key elements for storage, some of them are: 27, 37, 43, 78, 92.
Here consider a prime number > 20 i.e: 23
Now let us generate hash address.

Key	Division Method	Address
27	27 % 23 = 4	4
:	:	:
37	37 % 23 = 14	14
43	43 % 23 = 20	20
:	:	:
78	78 % 23 = 9	9
92	92 % 23 = 0	0

So now the physical memory or hash table:

Address	Key
00	92
01	
02	
03	
04	27
:	
09	78
:	
14	37
:	
20	43
21	
22	

Mid Square Method

→ Here, we can apply hash function to find the middle value of k^2 where k is the key element.

1) Find k^2
2) Identify middle value of k^2 as hash address

Consider an example:
Say there are 20 key elements for storage, some of them are:
27, 37, 43, 78, 92.

Key	Mid square Method	Address
27	$(27)^2 = 0729$	72
:		
37	$(37)^2 = 1369$	36
43	$(43)^2 = 1843$	84
:		
78	$(78)^2 = 6084$	08
92	$(92)^2 = 8464$	46

So now the physical memory or hash table:

Address	Key
00	
:	
08	78
:	
36	37
:	
46	92
:	
72	27
:	
84	43
99	

Folding Method

→ In this method we need to divide a key into number of partitions and then add them to identify address.

$H(K) = K_1 + K_2 + K_3 \dots K_n$

Example: Consider 30 numbers for storage, some of them are: 1234, 2345, 4567, 6789

Here we consider partition of 2 digits each.

key	Folding Method	Address
1234	12+34 = 46	46
2345	23+45 = 68	68
4567	45+67 = 112	12
6789	67+89 = 156	56

So, now the physical memory or hash table:

Address	key
00	
:	
12	4567
:	
46	1234
:	
56	6789
:	
68	2345
:	
99	