28/9

Data structures.

Easy to access

time ... space

Data structures - organization of Data in the primary memory for efficient access of Data. - accessing faster.

File structure - Organization of Data in the secondary memory.

a is an integer
↓ ↓
data type.

+ storage

Data structures.

When we know the type of the data we can able to know the range & operation that can be performed on that data.

Type :-

Collection of values

→ Boolean : true (or) False

→ Integers

Simple type

→ its value contains no subparts.

→ Integers

Aggregate / composite type :

combinations of two (or) more data type.

Ex: Bank Account :

→ name
→ address
→ account number
→ account balance

Date :

→ date
→ day
→ month
→ year

A data item is said to be a member of a type.

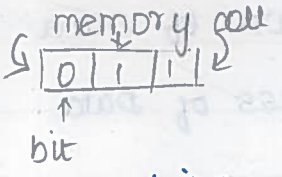
Computer memory & storage of data :

1 Bit ← 0 (or) 1

Bytes - 8 bit

Basic terminology :

Memory cell : circuit that stores 1-bit of information.



Memory word :- 8 to 64 bits.

Byte : a group of 8 bits.

Capacity (= Density)

- 4096 20-bit words

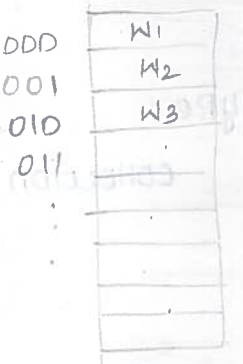
= 8192D bits = 4096 * 20 = 4k * 20

Address

Read Operation. fetch Operation

Write Operation store Operation

Addresses



Data type :

Character - 1 byte

Integer - 2 byte

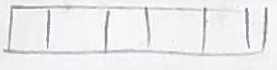
Float - 4 bytes

} depends on compiler it may vary.

bytes are consequent in memory.

1 bit - 2^1 = 2 values

Either 0 (or) 1



1 byte = 8 bit = 2^8 values.

(possible combinations)

Hexadecimal :

0-9, A-F } 16 values.

$16 = 2^{(4)}$ → 4 bits are required to store a single Hexadecimal value

Binary

0, 1 → 2 values → 1 bit

$2 = 2^{(1)}$

Little & Big Endian :

LSB - Lowest significant Bit / Byte

Big Endian Byte Order - The most significant byte (the "Big end") of the data is placed at the byte with the lowest address.

Little Endian Byte order - the least significant byte (the "little end") of the data is placed at the byte with the lowest address.

0xFF00 AA11 Little Endian Big Endian.

→

Address	contents
4000	FF
4001	00
4002	AA
4003	11

Address	contents
4003	11
4002	AA
4001	00
4000	FF

The rest of data is placed in order in the next three bytes in the memory.

Logical & Physical data :

User perspective & system perspective.

Ex : calendar date.

Logical : DD/MM/YYYY. ← user view

System ill taking care in memory allocation.

Physical : three integer (or) structure with three integer fields (or) an array with three integer elements. ↑ system view.

Abstraction : → not giving clear explanation of what it is?

↳ giving an overall picture.

→ keeping the interface common & hides the implementation.

03/10

Abstract Data Type (ADT) :

- type, operations are described.
- Hide the implementation.

ADT is the realization of a data type as a software component.

↳ data & operations.

The interface of the ADT is defined in terms of type and set of operation on that type.

ADT integer list :

- list of integer.

operation :

→ insert
→ delete
→ max
→ min
→ Average

But the implementation is Hidden.

Data structure :

A Data structure is a systematic way to organize data in computer main memory to enable efficient access of data.

DS is the implementation of ADT.

Data types.

ADT :

- Type
- Operation.

} Logical form.

⇓

Data structure :

- Storage space
- Subroutines.

} Physical form.

Choice of Data structure :

→ Analyse

→ Quantity

→ Select

Define ADT :

↓

(i) type of elements

(ii) Operations

Define Data structure :

(i) type of elements

(ii) operations

(iii) implementation.

Abstract Data type of Integer List :

Type of elements :-

List of integers can be stored.

Operations :-

→ Insert

→ Delete

→ Max

→ Min

→ Average

→ sorting of elements.

Problem solving:

Mathematical model ⇒ Abstract data type ⇒ Data structures

Informal Algorithm. ⇒ Pseudo Language Algorithm / program. ⇒ Program in C or Java or etc.,

Ex of collection of items:

visitors in Exhibition.

students in classroom

Members in a Family.

Friends in social media.

single data
↓
char, integer, float
↳ only we define have to be see the its ou available free space to store data.

But in the collection of items → we need to worry about the storing the entire collection in the list has to be stored in the contiguous memory locations.

visitors in Exhibition - Sets
↳ no other relationship between any other members in the set

linear collections

Pile of note book - stack. top pointer
↳ Last in First out

People in a ticket counter - Queue front & back pointer
↳ First in First out

Members in a Family - Tree
↳ Hierarchy of the items has to be noted.
one to many

Friends in social media - Graph
no formal way of connections.

Types of data structures :-

Primitive DS - fundamental data structure - defined on their own.

Compound DS - defined in terms of other data structures.

Example :-
Integer.
↑
it can have only integer value.
↑
defined on its own.

Example :-
pile of notebooks
↳ collection of int
↳ collection of string.
↓
so here collection is defined with other DS
↑
may/may not be contiguous

Linear DS :- access linear - one predecessor and one successor.

Example :- stack, queue.

Non Linear DS :- access not in linear - multiple predecessor / successor.

Example :- sets, tree, graphs.

Problem :-

→ A problem is a task to be performed

→ Input / matching outputs.

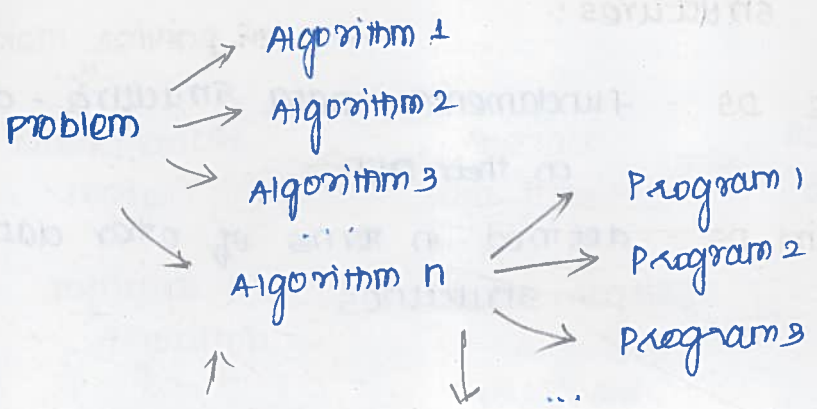
→ problem definition should not include any constraints on how the problem is to be solved.

↳ acceptable solutions.

↳ it must be run in a "reasonable" amount of time.

viewed as a function :-

matching between the input & output



Data structures concentrates on complexity (theoratically)

(i) time
(ii) space.

⇒ not bothering about the programing Language & configuration.

07/10

Asymptotic notation.

Algorithm :

An algorithm is a finite set of instructions which, if followed, accomplish a particular task.

It is a step-by-step unambiguous instructions to solve a given problem.

Main criteria :

- correctness
- Efficiency.

Computational thinking :

computational thinking problem solving process.

Elements :

- Decomposition (Breaking of problems)
- Pattern Recognition (Observing pattern)
- Abstraction (general principle)
- Algorithm Design. (step-by-step instructions)

Max & Min Algorithm : Given two numbers find max & min.

```
start
Get two number a and b
If a > b
    max = a
    min = b
else
    max = b
    min = a
end
```

Given three numbers find max and min.

```
start
Get three numbers a, b & c.
If a > b                                    see slide.
    If a > c
        max = a
    else
        max = c
else
    If b > c
    max = b
    else
        max = c
End.
```

Multiply two numbers using repeated addition.

```
Algorithm Multiply (A, B)
start
    C = 0
    While A is greater than 0
        C = C+B
        A = A-1
    End while
    Return C
    End.
```

Pow (a, n) without using exp function.

```
start
    pow = 1
    while n is greater than 0
        pow = pow * a
        n = n - 1
    end while
    return pow
end
```

2^3

$2 * 2 * 2$.

Iterative & Recursive Algorithms:

↓ set of instructions executed for n time

↳ Function calling itself for execution.

Ex: $fact(n) = n * (n-1)!$

$3! = 3 * (2!)$

$2 * (1!)$

1.

→ stores the Address & data using STACK Data Structures.

Ex:

```
Algorithm printNum(n)
start
    if (n==1) {
            print n
            return
        }
    else
        { print n
          printNum(n-1)
        }
end
```

printNum(5)

5
4
3
2
1.

Recursive algorithm for finding out n^{th} power of 2

```
Algorithm powNum(2,n):
                pow=1
start  if (n == 1)
        {
          pow = pow * 2
          return pow
        }
    else
        { pow = pow * powNum(2,n-1)
        }
end
```

check: ✓

2.

$2^3 =$

Properties of an Algorithm :-

* Finiteness
* Absence of ambiguity
* Definition of sequence
* Input / Output
* Feasibility.

If given down an algorithm and ask to check whether it satisfies all the properties.
If not so we have to change the Algo.

Efficiency measurement :-

Empirical comparison (run program & note down exe time)

Asymptotic Algorithm Analysis.

↓
It is not a proper way bec

21/10.

Array :

→ Finite ordered collection of homogeneous elements

→ static allocation.

→ operations :

* Access - sequential & random
* Insertion
* Deletion
* Searching

int data[10] ;
↑ ↑ ↑— no. of. elements
type name

Time complexity :
O(n)

↳ upper bound of the worst case.

maximum execution case.

(i) 1D array

```
a = {1,2,3,4,5}
for (i=0; i<n; i++)
    printf("%d", a[i]);
```

} 5 times
O(n) time complexity

(ii) 2D array

```
1 2 3
4 5 6
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        printf("%d", a[i][j]);
```

} 3 * 3
n * n
O(n²) time complexity.

Storage representation :

data = {1,2,3,4,5} int 2 bytes

Base address: 1004

data[0]	[1]	[2]	[3]	[4]					
1	2	3	4	5					
1004 1005	06	07	08	09	10	11	12	13	14

Physical address : single dimensional array with each & every byte contains unique address.

Physical address: (1D Array)

Addressing Function } : Address(data[i]) = base address + (i * element_size)

2D Array:

Logical representation - rows, columns

Physical/storage representation - single dimensional memory.

1	2	3
4	5	6
7	8	9

Row major order

00	01	02	10	11	12	20	21	22
1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8

rows :

Column major order

columns :

1	4	7	2	5	8	3	6	9

↓

2 index - row index & column index.

i j

Physical Address: (2D Array)

Row major order :

Addressing Function } : Address(data[i][j]) = baseaddress + element_size (i * cols + j)

(i * cols + j) ← preceding. current row

Column major order :

Addressing function } : Address(data[i][j]) = base address + element_size (j * rows + i)

(j * rows + i) ← preceding current

26/10 :

Storage representation of 3D arrays :

store layer by layer

→ row/column major order.

Row major order :

Addressing function } : Address(data[i][j][k]) = base address + size(k * rows * cols + i * cols + j)

k → depth / slice

Column major order

Addressing function :- Address (data [i][j][k]) =

base address + size((k * rows * cols) + (j * rows + i))

Special Matrices :-

Diagonal Matrix :

(0,0)

1 0 0 0
0 2 0 0
0 0 3 0
0 0 0 4

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

→ storing only non zero elements.

Non zero elements :- (i=j)

2D is converted into 1D

0	1	(0,0)
1	2	(1,1)
2	3	(2,2)
3	4	(3,3)

Addressing Function :-

Address (DATA[i][j]) = base address + i * elements_s ← if i=j

otherwise

DATA[i][j] = 0

Lower triangular Matrix :

(0,0)

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

(1,0) (1,1) (2,0) (2,1) (2,2) (3,0) (3,1) (3,2) (3,3)

↑ non zero

elements above the main diagonal are zero

(i) relationship between row & column.

(ii) How many non zero elements nxn size.

(iii) For an efficient storage if only the non zero elements are stored → Addressing function

(i) relationship between rows & columns

row → i , column → j

Lower triangular } Non-zero elements ← $j <= i$ (or) $i >= j$

(ii) No. of. non zero elements stored of size nxn

n=1 ⇒ 1 n=4 ⇒ 10.

n=2 ⇒ 3 n=5 ⇒ 15

n=3 ⇒ 6

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

(iii) Addressing function for non-zero elements.

(0,0)	1	0
(1,0)	2	1
(1,1)	3	2
(2,0)	4	3
(2,1)	5	4
(2,2)	6	5
(3,0)	7	6
(3,1)	8	7
(3,2)	9	8
(3,3)	10	9

Address (2,2) :

No. of. elements in row 0 & 1 = 1+2 = 3

No. of. preceding non-zero elements in row 2 } = 2

offset = 3+2 = 5

Address (3,3) =

No. of. elements in row 0, 1, 2 = 1+2+3 =

No. of. preceding non-zero elements in row 3 } = 3

offset = 6+3 = 9

Addressing Functions :

Address (data[i][j]) = Base address + element size ($\sum_{k=0}^{i} k + j$) ∀ $i >= j$

preceding row → current row

(or)

Base_address + element size ($\frac{i(i+1)}{2} + j$)

otherwise

data[i][j] = 0

Upper triangular Matrix:

(0,0)	(0,1)	(0,2)	(0,3)
1	2	3	4
	(1,1)	(1,2)	(1,3)
0	5	6	7
		(2,2)	(2,3)
0	0	8	9
0	0	0	10
			(3,3)

(i) relationship between rows & columns

row → i column → j

Upper triangular { Non. zero elements ← $i <= j$ (or) $j >= i$

(ii) No. of. non zero elements stored of size n×n

$\frac{n(n+1)}{2} \Rightarrow$ Eg: $n=4 \Rightarrow \frac{4(4+1)}{2}$

$= 20/2 = 10$ ✓

(iii) addressing function for non zero elements.

Lower triangular

(i) 0th row has no preceding rows } ⇒ so no of non zero elements } = 0.

(ii) 1st row has 1 non zero element in its preceding row
(0+1)

(iii) 2nd row has 3 non zero element in its preceding row.
↳ (0+1+2)

(iv) 3rd row has 6 non zero elements in its preceding row.
↳ (0+1+2+3)

In general ith row

$= \sum_{k=0}^{i} k \rightarrow \frac{i(i+1)}{2}$ → no of non zero elements in its preceding rows.

Upper triangular Matrix:-

(0,0)	(0,1)	(0,2)	(0,3)
1	2	3	4
	(1,1)	(1,2)	(1,3)
0	5	6	7
		(2,2)	(2,3)
0	0	8	9
			(3,3)
0	0	0	10

(i) relationship between rows & columns

row → i column → j

upper triangular { Non. zero elements ← i<=j (or) j>=i

(ii) No. of. non zero elements stored of size n×n

$$\frac{n(n+1)}{2} \Rightarrow$$

Eg: $n=4 \Rightarrow \frac{4(4+1)}{2}$

$= 20/2 = 10$ ✓

(iii) Addressing function for non zero elements.

(0,0)	1	0
(0,1)	2	1
(0,2)	3	2
(0,3)	4	3
(1,1)	5	4
(1,2)	6	5
(1,3)	7	6
(2,2)	8	7
(2,3)	9	8
(3,3)	10	9

n = no of rows / columns.

preceding row = n-i

Addressing Functions:-

Address (data[i][j]) = if i<=j

Base address +

element_size ($\sum_{k=0}^{i}$ (rows - k) + (j-i))

otherwise

data[i][j] = 0.

(1) **k-Band matrices :** $n \times n$

→ Elements on the main diagonal, one diagonal above the main diagonal & below the main diagonal are non-zero elements.

→ All other are zero.

(2) **Symmetric matrices :**

$$A[i][j] = A[j][i]$$

where $i \neq j$

Answers :

1. k-Band Matrix : $(n \times n) \rightarrow (k \times k)$

(0,0) 1	(0,1) 2	0	0
(1,0) 6	(1,1) 3	(1,2) 4	0
(2,1) 0	(2,2) 8	(2,3) 7	9
0	0	(3,2) 10	(3,3) 11

No of non zero elements at

0th row → 2

1st row → 3

2nd row → 3

3rd row → 2

No of preceding non zero element at

0th row → 0

1st row → 2

2nd row → 5

3rd row → 8

In general no of. preceding non-zero elements at ith row $= 2 + (i-1)3 = 2 + 3i - 3 = 3i - 1$

(0,0)	1	0
(0,1)	2	1
(1,0)	6	2
(1,1)	3	3
(1,2)	4	4
(2,1)	8	5
(2,2)	7	6
(2,3)	9	7
(3,2)	10	8
(3,3)	11	9

Address (Data[i][j]) = Base address + element_size ($3i-1$ + $j-i+1$)

↑ no of preceding non zero

↳ column

$$= \text{Base address} + \text{element\_size}(2i + j)$$

if $i-1 \leq j \leq i+1$

else return 0.

2. symmetric matrix :

15 . unique elements

0,0	0,1	0,2	0,3	0,4
1	2	3	4	5
1,0	1,1	1,2	1,3	1,4
2	4	6	8	10
2,0	2,1	2,2	2,3	2,4
3	6	9	12	15
3,0	3,1	3,2	3,3	3,4
4	8	12	16	20
4,0	4,1	4,2	4,3	4,4
5	10	15	20	25

(i) relationship between rows & columns

$A[i][j] = A[j][i]$

$\forall\ i >= j$

(ii) Addressing Functions.

It is similar to lower triangular matrix, in storing a one single part of symmetric matrix.

	(0,0)	1	0
(0,1)	(1,0)	2	1
	(1,1)	4	2
(0,2)	(2,0)	3	3
(1,2)	(2,1)	6	4
	(2,2)	9	5
(0,3)	(3,0)	4	6
(1,3)	(3,1)	8	7
(2,3)	(3,2)	12	8
	(3,3)	16	9
(0,4)	(4,0)	5	10
(1,4)	(4,1)	10	11
(2,4)	(4,2)	15	12
(3,4)	(4,3)	20	13
	(4,4)	25	14

Addressing Function =

Address (Data [i][j]) =

Base_address +

element_size $\left(\frac{i(i+1)}{2} + j \right)$

$\forall\ i >= j$

Otherwise $j > i$

Address (Data[i][j]) =

Base address +

element_size $\left(\frac{j(j+1)}{2} + i \right)$

3. Toeplitz matrix

(0,0)	(0,1)	(0,2)	(0,3)
1	2	3	4
(1,0)	(1,1)	(1,2)	(1,3)
5	1	2	3
(2,0)	(2,1)	(2,2)	(2,3)
6	5	1	2
(3,0)	(3,1)	(3,2)	(3,3)
7	6	5	1

n x n.

n be the dimension.

0	1	(0,0), (1,1), (2,2), (3,3)
1	2	(0,1), (1,2), (2,3)
2	3	(0,2), (1,3)
3	4	(0,3)
4	5	(1,0), (2,1), (3,2)
5	6	(2,0), (3,1)
6	7	(3,0)

⇒ $j \geq i$

$= j - i$

⇒ otherwise

$= (n-1) + (i-j)$

(2,0) ⇒

$= (4-1) + (2-0)$

$= (3) + (2)$

$= 5$

Storage representation of toeplitz matrix:

Address (Data[i][j]) = if $j \geq i$

Baseaddress +

element_size (j-i)

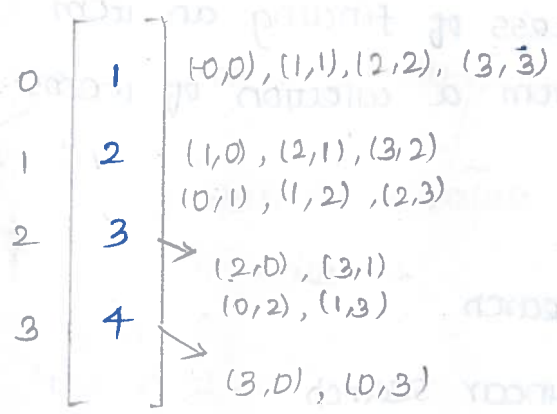
else $j < i$

baseaddress +

element_size ((n-1) + (i-

4. toeplitz symmetric matrix.

(0,0)	(0,1)	(0,2)	(0,3)
1	2	3	4
(1,0)	(1,1)	(1,2)	(1,3)
2	1	2	3
(2,0)	(2,1)	(2,2)	(2,3)
3	2	1	2
(3,0)	(3,1)	(3,2)	(3,3)
4	3	2	1



storage representation of symmetric toeplitz matrix:

Address(data [i][j]) = Base address + element_size (|i-j|)

↳ absolute value / modulus value

+(i-j))

Peak Element :

An element is called a peak element if its value is not smaller than the value of its

Searching.

Linear search Binary search Pattern search.

Searching is the process of finding an item with specified properties from a collection of items.

Types of searching :

- Unordered Linear search
- sorted / ordered Linear search
- Binary search
- Symbol Tables & Hashing
- Pattern / String searching Algorithms.

Linear search : search a key value in a 1D array.

```
start
    read no of elements as n
    for i=0 to n do
      begin
          Arr[i]=read value;
      end
    read key
    count = 0
    for i=0 to n do
    begin
          if Arr[i] == key;
             count=1.
    end
```

```
if count == 1
    print("Element present")
else
    print("Element not present")
End.
```

Unordered Linear search :

```
Algorithm UnorderedLinearSearch(data, n, value)
Begin
    for i = 0 to n-1 do
    begin
        if data[i] == value
            return i
    end
    return -1.
End
```

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

complexity :

Time complexity : O(n)

Space complexity : O(1)

Tracing :

data | 17 | 16 | 1 | 5 | 20 | 14 | 3 |
0 1 2 3 4 5 6

n = 7 value = 20

Start

i = 0 — if data[0] == value : 17 == 20 × false

i++ → i = 1

i = 1 — if data[1] == value : 16 == 20 × false

i++ → i = 2

i = 2 — if data[2] == value : 1 == 20 × false

i++ → i = 3

i = 3 — if data[3] == value : 5 == 20 × false

i++ → i = 4

i = 4 — if data[4] == value : 20 == 20 ✓ true

return 4

End

O/p :

4th position.

Implementation in python :-

```
def UnorderedLinearsearch(data, value):
    for i in range(len(data)):
        if data[i]==value:
            return i
    return -1.
```

Ordered Linear search

Algorithm Ordered Linear Search (data, n, value):

```
Begin
    for i=0 to n-1 do
    begin
        if data[i]==value
            return i
        else if data[i] > value
            return -1
    end
    return -1
End
```

Best case : O(1)

Worst case : O(

complexity :

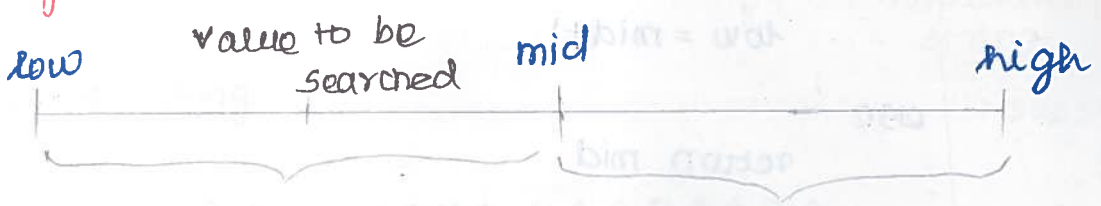
Time complexity : O(n)

space complexity : O

Implementation in python :

```
def OrderedLinear search(data, value):
    for i in range(len(data)):
        if data[i]==value:
            return i
        elif data[i]> value:
            return -1
    return -1
```

Binary search :



2	4	6	8	10	12	14	16	18	20
0	1	2	3	4	5	6	7	8	9

case(i) :

value == data [mid]

return mid as position .

case(ii) :

value < data [mid]

high = mid - 1

$$mid = \left\lfloor \frac{(low + high)}{2} \right\rfloor$$

goto case(i)

Best case → input that got at the minimu

Best case : O(1).

case(iii) :

value > data [mid]

low = mid + 1

$$mid = \left\lfloor \frac{(low + high)}{2} \right\rfloor$$

goto case(i)

Iterative Algorithm :

Algorithm Iterative Binary search (data, n, value) :

Begin

low = 0

high = n - 1

while (low <= high) {

mid = (high + low)/2

if (data [mid] > value)

high = mid - 1

```
        else if (data[mid] < value)
                low = mid+1
        else
                return mid

        return -1

End .
```

Recursive Algorithm :

```
Algorithm RecursiveBinarysearch (data, h, low, high)
Begin
        if low > high
                return -1
        else
                mid = (low+high)/2
                if value == data[mid]
                        return mid
                else if value > data_[mid]
                        return RecursiveBinarysearch(data, value,
                                                mid+1, high)
                else
                        return RecursiveBinarysearch (data, value,
                                                low, mid-1)
End
```

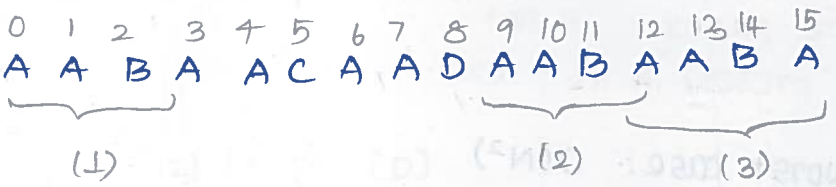
Pattern searching Algorithm

Patter searching:

string
↳ substring search

Text : AABA ACAA DAABAABA.

Pattern : AABA



Naive Pattern searching:

(all possibility checking)

i=0 to len(Text) ⇒ n

j=0 to len(pattern) → pattern can be of length len(Text) ⇒ n

Worst case Time complexity } O(n²)

Algorithm NaivePatSearch (pat, txt)

Begin

```
M = length (Pat)
N = length(txt)
[check characters in pat one by one]
for i=0 to N-M {
    j=0
    while (j<m) {
        if (txt[i+j] != pat[j])
            break
        j+=1
    }
    if (j==M)
        print(" Pattern found at index ",i)
}
```

End

complexity :-

Time complexity : $O(N^2)$

Auxiliary space : $O(1)$ doesn't require any additional space, whose size is directly proportional to the size N.

Best case : $O(N)$

First character of the pattern is not present in the text.

Worst case :- $O(N^2)$

(1) All characters of the text and pattern are the same.

txt[] = "AAAAAAAAAA" pat[] = "AA"

(2) Only the last character is different

txt[] = "AAAAAAAAAB" pat[] = "AAB".

Text :- A A B A A C A A D A A B A A B A len(text) = 16

pattern :- X A B A len(pattern) = 4

i=0, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

i=1, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

i=2, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

i=3, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

i=4, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

. . . .

i=12, j=0 : A A B A A C A A D A A B A A B A } X
X A B A

Algorithm tracing :

M = 4
N = 16
i = 0 to 12

A != X

i=0 j=0 txt[0] != pat[0] break ✓

i=1 j=0 txt[1] != pat[0] break ✓

. . . . .

. . . . .

i=12 j=0 txt[12] != pat[0] . . .

KMP Pattern searching :

↳ Knuth Morris Pratt (KMP) pattern searching.

Reduces worst case complexity to O(n)

Idea : skip prefix of the pattern.

this requires to know how many characters can be skipped.

Step 1 :

txt : AAAA ABAAABA

pat : AAAA

Step 2 :

txt : AAAAA BAAABA

pat :

Tracing Algorithm Naive pat Search (pat, txt) :

Input : pat = AABA

txt = AABAACAADAABAABA.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Begin :

M = 4

N = 16

for i = 0 to N-M
16-4 = 12

i = 0 to 12

⇒ i = 0

→ j = 0 0<4 ✓

while (j<m) A != A ✗

if txt[0+0] != pat[0]

j += 1 ⇒ j = 1

→ j = 1 1<4 ✓

while (j<m) A != A ✗

if txt[0+1] != pat[1]

j += 1 ⇒ j = 2

→ j = 2 2<4 ✓

while (j<m) B != B ✗

if txt[0+2] != pat[2]

j += 1 ⇒ j = 3

→ j = 3 3<4 ✓

while (j<m) A != A ✗

if txt[0+3] != pat[3]

j += 1 ⇒ j = 4

→ j = 4 4<4 ✗

while (j<m)

4 == 4 ✓

if j == m

print "pattern found at index 0".

⇒ i = 1 →

⇒ i = 2 →

→ i = 3 →

⇒ i = 4 → j =

→ j

⇒ i=1

→ j=0 0<4 ✓

while j<m A!=A ✗

if txt[1+0] != pat[0]

j=j+1 ⇒ j=1

→ j=1 1<4 ✓

while j<m B != A ✓ true.

if txt[1+1] != pat[1]

break ✓

⇒ i=2

→ j=0 0<4 ✓

while j<m B != A ✓ true

if txt[2+0] != pat[0]

break ✓

⇒ i=3

→ j=0 0<4 ✓

while j<m A != A ✗

if txt[3+0] != pat[0]

j+=1 ⇒ j=1

→ j=1 1<4 ✓

while j<m A!=A ✗

if txt[3+1] != pat[0]

j+=1 ⇒ j=2

→ j=2

while 2<4 ✓ C!=B ✓

if txt[3+2] != pat[0]

break ✓

⇒ i=4

→ j=0 0<4 ✓

while j<m A!=A ✗

if txt[4+0] != pat[0]

j+=1 ⇒ j=1

→ j=1 while 1<4 ✓ C!=A ✓

if txt[4+1] != pat[0] break ✓

⇒ i=5

→ j=0

while 0<4 ✓ C != A ✓

if txt[5+0] != pat[0]

break ✓

⇒ i=6

→ j=0 ✓

while 0<4 A != A ✗

if txt[6+0] != pat[0]

j+=1 ⇒ j=1

→ j=1

while 1<4 ✓ A != A ✗

if txt[6+1] != pat[0]

j+=1 ⇒ j=2

→ j=2

while 2<4 ✓ D != B ✓

if txt[6+2] != pat[0]

break ✓

⇒ i=7

→ j=0

while 0<4 ✓ A != A ✗

if txt[7+0] != pat[0]

j+=1 ⇒ j=1

→ j=1

while 1<4 ✓ D != A ✓

if txt[7+1] != pat[0]

break ✓

⇒ i=8

→ j=0

while 0<4 ✓ D != A ✓

if txt[8+0] != pat[0]

break ✓

⇒ i=9

→ j=0

while 0<4 ✓ A != A ✗

if txt[9+0] != pat[0]

j+=1 ⇒ j=1

→ j=1

while 1<4 ✓ A != A ✗

if txt[9+1] != pat[0]

j+=1 ⇒ j=2

→ j=2

while 2<4 ✓ B != B ✗

if txt[9+2] != pat[0]

j+=1 ⇒ j=3

→ j=3

while 3<4 ✓ A != A ✗

if txt[9+3] != pat[0]

j+=1 ⇒ j=4

→ j=4

while 4<4 ✗ false

4==4 ✓

if j==m

print "pattern found at index 9"

⇒ i=10

→ j=0

while 0<4 ✓ A != A ✗

if txt[10+0] != pat[0]

j+=1 ⇒ j=1

→ j=1

while 1<4 ✓ B != A ✓

if txt[10+1] != pat[0]

break ✓

⇒ i=11

→ j=0

while 0<4 ✓ B != A ✓

if txt[11+0] != pat[0]

break ✓

⇒ i=12

→ j=0

while 0<4 ✓ A != A X

if txt[12+0] != pat[0]

j+=1 ⇒ j=1

→ j=1

while 1<4 ✓ A != A X

if txt[12+1] != pat[1]

j+=1 ⇒ j=2

→ j=2

while 2<4 ✓ B != B X

if txt[12+2] != pat[2]

j+=1 ⇒ j=3

→ j=3

while 3<4 ✓ A != A

if txt[12+3] != pat[3]

j+=1 ⇒ j=4

→ j=4

while 4<4 ✗ false

if j==m 4==4 ✓

print ("pattern found at index 12")

output :

pattern found at index 0

pattern found at index 9

pattern found at index 12

Pre-processing : calculation of LPS :

LPS - Longest proper prefix which is also suffix.

Example :

proper prefixes are " ", "A" and "AB"

suffixes of the string are } " ", "C", "BC" and "ABC".

LPS[i] = length of the longest

Pattern : AAAA

Pat[0...0] = "A"

[proper prefix " " is a suffix length is 0]

LPS[0] = 0

Pat[0...1] = "AA"

[proper prefix " ", "A" is a suffix, length 1]

LPS[1] = 1

similarly:

Pat[0...2] ⇒ LPS[2] = 2

Pat[0...3] ⇒ LPS[3] = 3

⇒ LPS[Pat] = [0, 1, 2, 3].

calculation of LPS :

1. pattern " ABCDE "

pat [0...0] = " A "

prefix & suffix : " " → length = 0

LPS [0] = 0

pat [0...1] = " AB "

prefix : " ", " A "
suffix : " ", " B " → length = 0

LPS[1] = 0

pat [0...2] = " ABC "

prefix : " ", " A ", " AB "
suffix : " ", " C ", " BC " → length = 0

LPS[2] = 0

pat [0...3] = " ABCD "

prefix : " ", " A ", " AB ", " ABC "
suffix : " ", " D ", " CD ", " BCD " → length = 0

LPS[3] = 0

pat [0...4] = " ABCDE "

prefix : " ", " A ", " AB ", " ABC ", " ABCD "
suffix : " ", " E ", " DE ", " CDE ", " BCDE " → length =

LPS[4] = 0

LPS [] = [0, 0, 0, 0, 0]

2. pattern : " AAB AAC AAB AA

pat [0...0] = " A "

prefix : " " → length = 0

pat [0...1] = " AA "

prefix : " ", " A " → length = 1 ✓

pat [0...2] = " AAB "

prefix : " ", " A ", " AA " → length = 2 ✓

Note :

We have to ch
the longest pref
present at s
or not

pat[0..3] = "AABA"

prefix: "", "A", "AA", "AAB" → length = 2

✓ ✗

pat[0...4] = "AABAA"

✓

prefix: "", "A", "AA", "AAB", → length = 2

"AABA" ✗

✗

pat[0...5] : "AABAAC"

prefix: "", "A", "AA", "AAB", → length = 5

"AABA", "AABAA"

✓

pat[0...6] : "AABAACA"

prefix: "", "A", "AA", "AAB", → length = 5

"AABA", "AABAA", "AABAAC"

✓ ✗

pat[0...7] : "AABAACAA"

prefix: "", "A", "AA", "AAB", "AABA", → length = 5

"AABAA", "AABAAC", "AABAACA"

✓ ✗ ✗

pat[0...8] : "AABAACAAB"

prefix: "", "A", "AA", "AAB", "AABA",

"AABAA", "AABAAC", "AABAACA", → length = 5

✓ ✗

"AABAACAA"

✗

pat[0...9] : "AABAACAABA"

prefix: "", "A", "AA", "AAB", "AABA",

"AABAA", "AABAAC", "AABAACA",

"AABAACAA", "AABAACAAB"

AAAC

KMP Algorithm :

1. calculate lps
2. Let N be the length of txt and M be the length of pat
3. set i=0 and j=0 [i for index in txt and j for index in pat]
4. Repeat while (N-i) >= (M-j) Begin

If txt[i] and pat[j] match
increment both i and j

If j=M,
pattern is found at index i-M
Reset j = lps[j-1]

If txt[i] and pat[j] do not match and j>0
Reset j = lps[j-1]

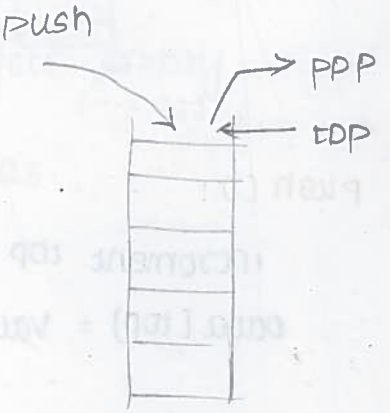
If txt[i] and pat[j] do not match and j=0
Increment i

C02. STACKS

→ subcase of sequential data structures - limited data structures.

Arrays : Random access

Lists : sequential access



Last In First Out (LIFO) List.

Operations:

Push, Pop and Peek

isempty

Applications:

Reverse a word

solve Maze game

space for parameters and local variables is created internally using a stack.

compiler's syntax check for matching braces is implemented by using stack.

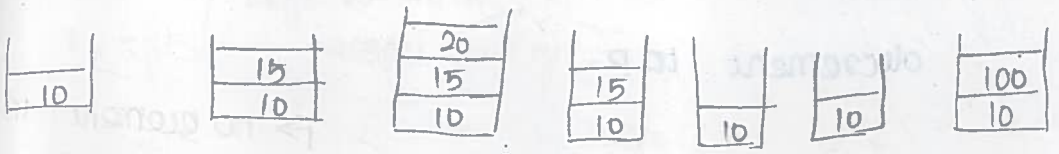
support for recursion.

push → inserting an element into stack at top

pop → removes an element at top from stack

peek → returns an element at top from stack.

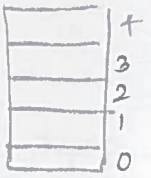
push(10), push(15), push(20), pop, pop, peek, push(100)



20 20, 15 10

Implementation of stack in Array :

data[5]



top → top element
top → location where element can be placed

N → size of Array

top = -1

push() :

increment top
data[top] = value

If top == N-1 → may leads to over flow condition
stack Full

else :

top ++
data[top] = value.

1. check for overflow
2. If not overflow then
   increment top
   add element into stack.

pop() :

print data[top]
decrement top

If top == -1 → may leads to under flow condition
stack empty

else :

print data[top]
top --

1. check for underflow
2. If not underflow then
   print Data[top]
   decrement top.

peek() :

print Data[top]

If top == -1 → no element in stack.
stack empty

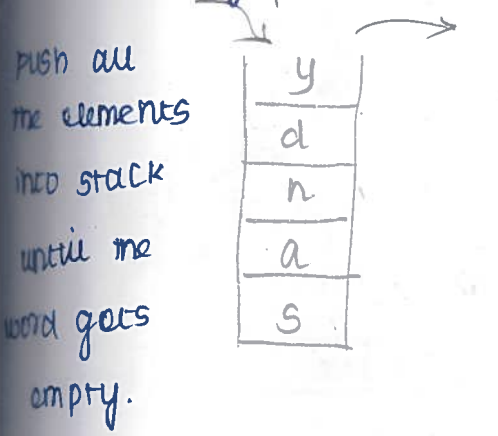
else

print Data[top]

Application :

Reversing of word :

→ sandy push pop



Infix Expression :

↳ operators present inbetween operands

a + b * c

a = 2 b = 3 c = 4

2 + 3 * 4

12

14

Infix to postfix :

a+b → ab+

a + b * c → abc*+

(i)

Precedence table :

↑ Right to left associativity

*, / } Left to right associativity

+, -

1. a + b*c *d - f

bc*

abc*d*

abc*d*+

abc*d*+f-

2. $a + b/c - d*e - f$

$bc/ \quad de*$

$abc/+$

$abc/+de*-$

$abc/+de*-f-$

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

$bc+ \quad df/$

$bc+df/-$

$a\,bc+df/-*$

$abc+df/-*e*$

4. $a + (b+c - (b*d) + e) - f$

$bc+ \ominus \quad bd*$

$bc+bd*-\oplus e$

$a \oplus \quad bc+bd*-e+$

$bc+bd*-e+a+ \quad \ominus f$

$bc+bd*-e+a+f-$

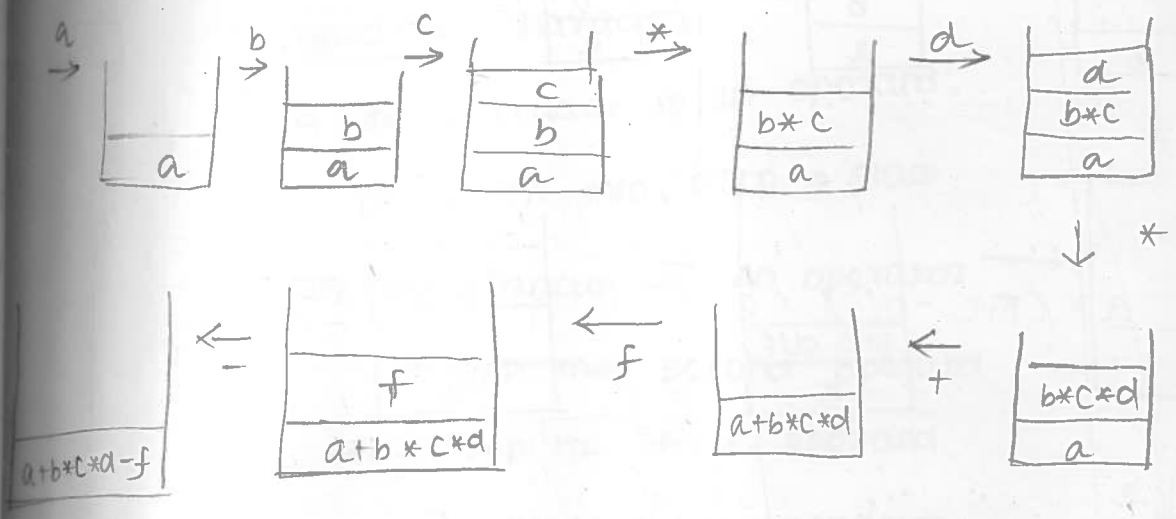
$abc+bd*-e++$

$\ominus f$

$abc+bd*-e+f+-$

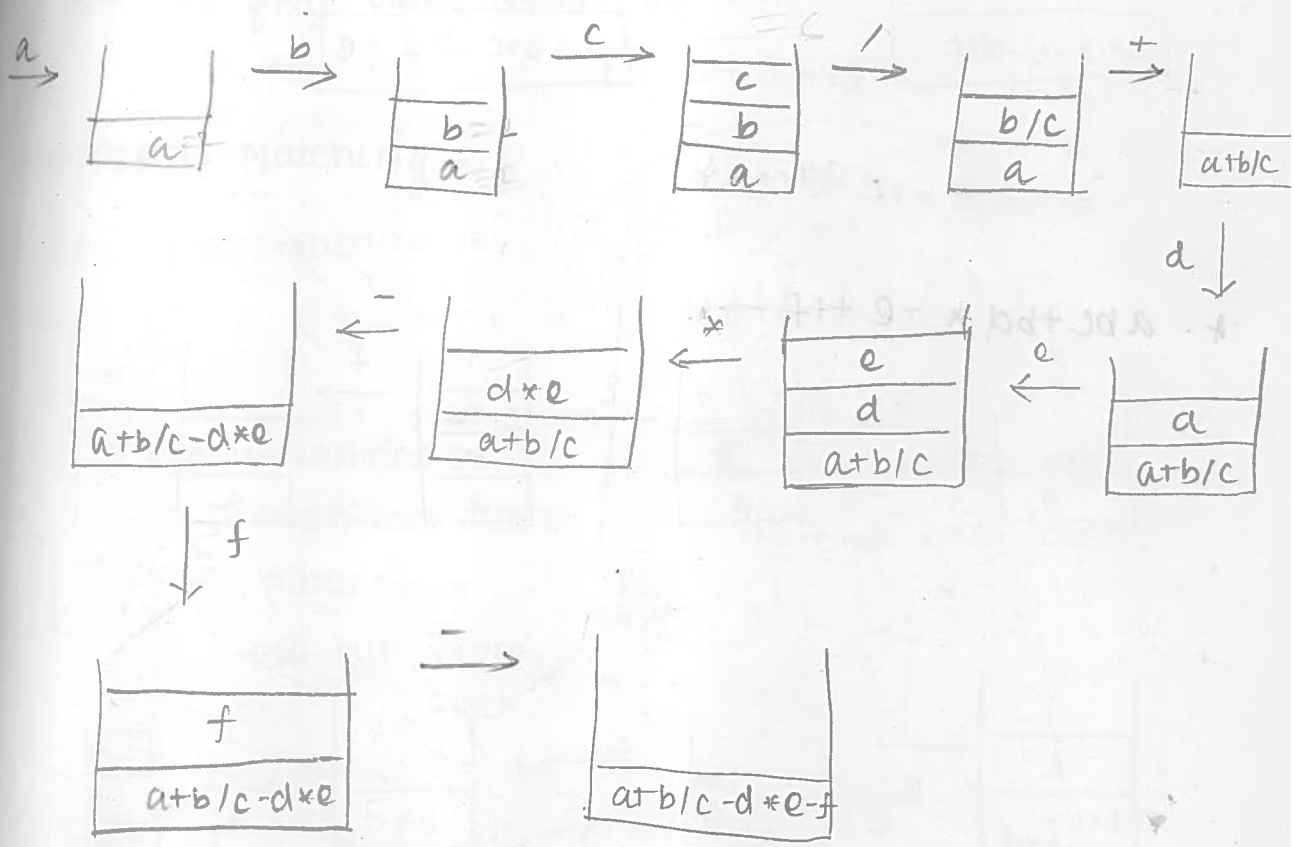
Stack Evaluation of Postfix Expression:

1. abc*d*+f-



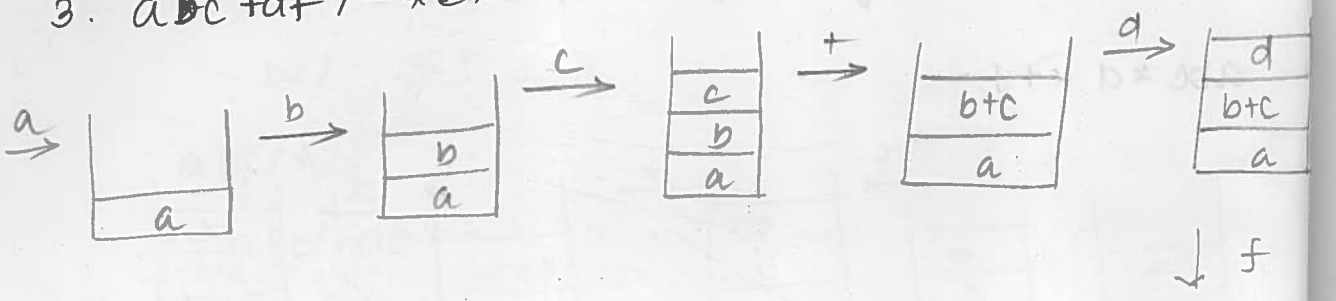
⇒ a+b*c*d-f

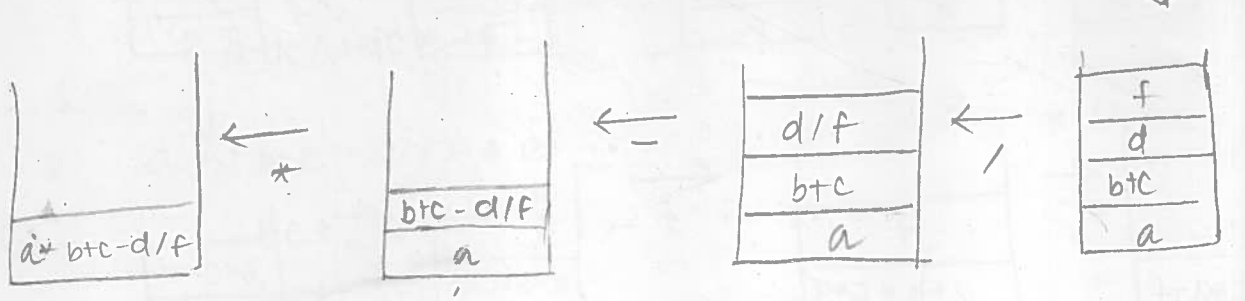
2. abc/+de*-f-

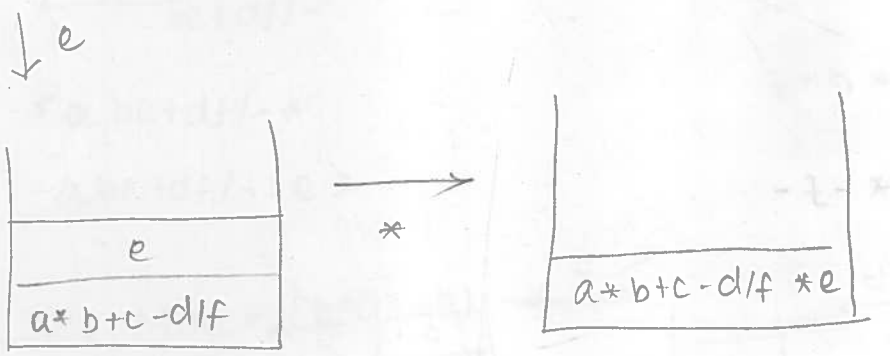


⇒ a+b/c-d*e-f

3. a b c + d f / − * e *

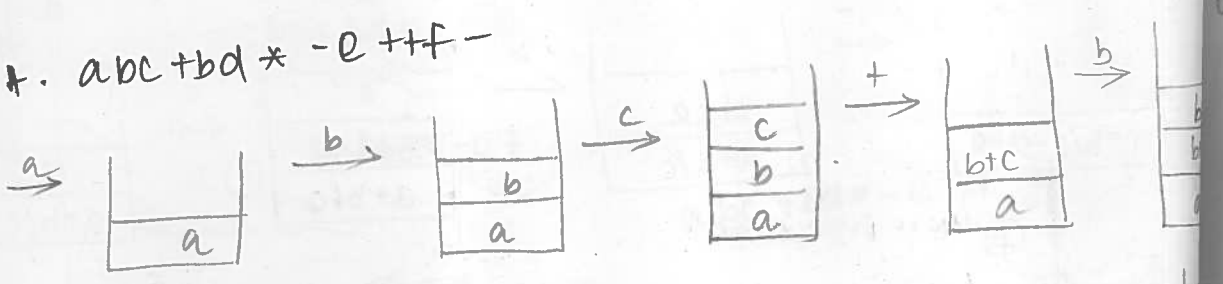


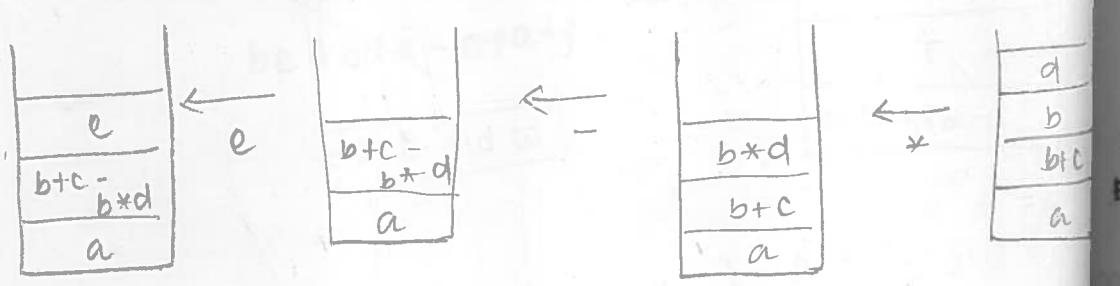


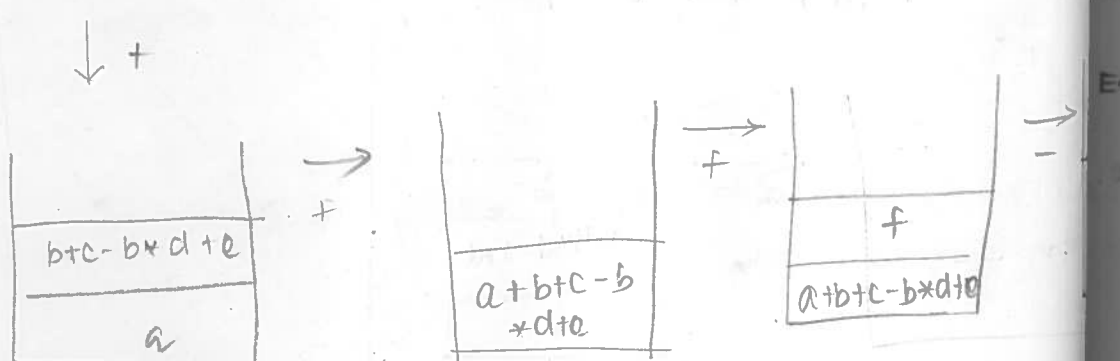


⇒ a * b + c - d / f * e

4. a b c + b d * - e + + f -







⇒ a + b + c - b * d + e - f

Evaluating a postfix expression :-

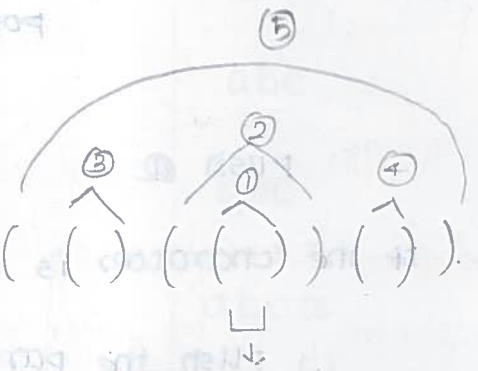
while characters remain in the postfix string.

1. read a character
2. If the character is an operand.
   → push into onto a stack.
3. If the character is an operator
   → pop the second operand
   → pop the first operand
   → perform the operation
   → push the result.

pop the final value from the stack.

Parenthesis Matching :-

stack datastructure is used.



(→ open parenthesis
push into stack

) → close parenthesis
pop out from stack

① most recent open matches first close.

⑤

Excess of closing parenthesis :-

trying to popout element from Empty stack.

Excess of open parenthesis :-

If the stack is not empty after reaching the end of string.

18/11

Infix to postfix notation :

Algorithm :

While characters remain in the infix string.

1. read the next character in the infix string
2. if the character is an operand,
   (i) append the character to the postfix expressing.
3. If the character is an operator @
   (i) while the stack is non empty and an operator of greater or equal priority is on the stack
   pop the stack and append the operator to the postfix
   (ii) push @
4. If the character is a left paranthesis '('
   (i) push the paranthesis onto the stack
5. If the character is a right paranthesis ')'
   (i) while the top of the stack is not a left paranthesis '('
   pop the stack and append the operator to the postfix expression
   (ii) pop the stack '(' and discard the left paranthesis.

pop any remaining items on the stack and append postfix expression

Example :-

Infix :- a + (b+c * a +d) / c .

Postfix :- abc a* +d +c/+ .

Infix Expression	Stack_content	Postfix Expression
a+(b+c *a+d)/c ↑		a
+(b+c *a+d)/c ↑	+	a
(b+c * a +d) /c ↑	+ (	a
b+c * a +d)/c ↑	+(	ab
+c * a+d)/c ↑	+(+	ab
c *a+d) /c ↑	+(+	abc
* a+d)/c ↑	+(+ *	abc
a+d) /c ↑	+(+ *	abca
+d) /c ↑	+(+ * → POP +	abca *
	+(+ → POP +	abca* +
d) /c ↑	+(+	abca*+d
)/c ↑	+ (+ POP until '('	abca*+d+
/c ↑	+ /	abca *+d +
c ↑	+/	abca *+d + c
Infix exp empty	POP all the values until the stack gets empty.	abca*+d+c/+

Paranthesis Matching :-

Algorithm paranthesesCheck (infix) :-

Begin

1. create stack S
2. balanced = true
3. index = 0
4. while index < len(infix) and balanced == True

   (i) symbol = infix[index]

   (ii) if symbol == '('

   push (S, symbol)

   else if symbol == ')'

   If isEmpty(S)

   balanced = False

   else

   pop(S)

   (iii) index = index + 1

5. if balanced and isEmpty(S)

   return true

   ↳ Balanced paranthesis

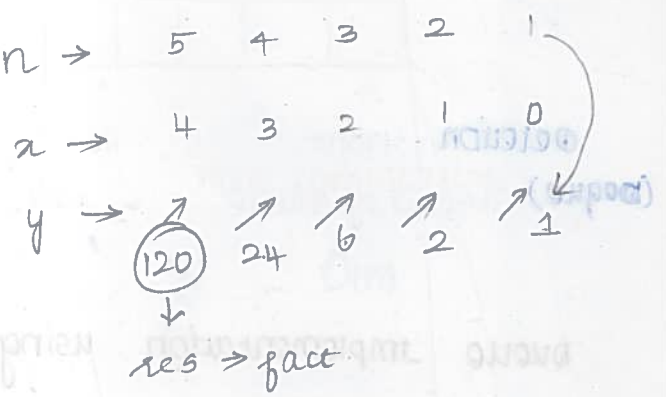
   else

   return False.

   ↳ paranthesis is unbalanced

Recursive :-

```
Algorithm  factorial (n) :-
begin  if (n==0) then
           fact = 1
       else {
           x = n-1
           y = factorial (x)
           fact = n * y
       }
   return fact
End.
```



Time Complexity of stack :-

When size of stack is of size 'n'.

Insertion - O(n)

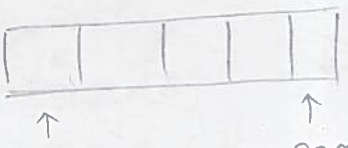
Deletion - O(n)

Searching for an element - O(n)

21/11/22

Queues :

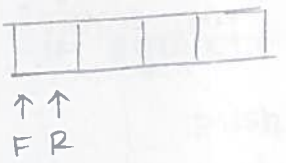
FIFO - First in First Out.



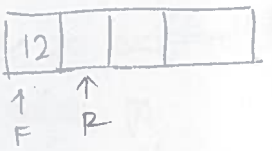
Deletion - front (Dequeue) Head of Queue

Rear - Insertion (Enqueue) tail of Queue

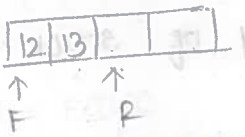
Queue Implementation using Array :



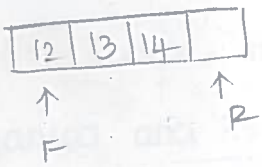
Insert 12



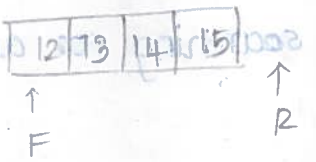
Insert 13



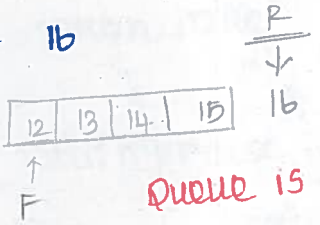
Insert 14



Insert 15



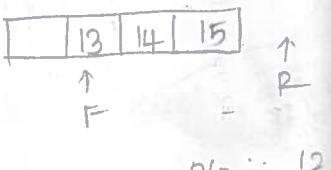
Insert 16



Queue is Full

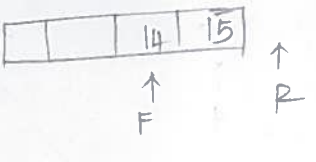
When rear pointer reaches the size of Queue.

Delete



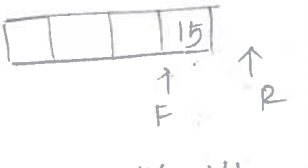
O/p :- 12

Delete



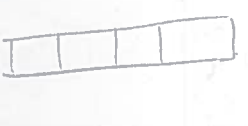
O/p :- 13

Delete



O/p :- 14

Delete



Queue is Empty.

When Front >= rear po
then the Queue is Empty

Algorithm insertQueue (Queue, n, Rear, item)

Begin

if (Rear == n)

then Queue Full

else

Queue [Rear] = item

Rear = Rear + 1

End

Time complexity:

O(n)

Algorithm deleteQueue (Queue, *, Rear, Front):

Begin

if (Front >= Rear) (OR) ==

then Queue Empty

else

print Queue [Front]

Front = Front + 1

End

Time complexity:

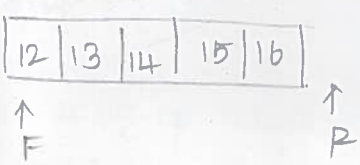
O(n)

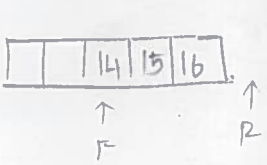
Limitation of Linear Queue:

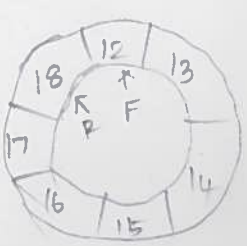
In Linear Queue After the Queue is Full, we try to delete 1 or 2 elements from Queue. It implies that we have space in the Queue. But the rear pointer is facing at n^{th} index.

So we are not able to insert an element inside the queue, although we have space.

To overcome this we go for circular Queue.

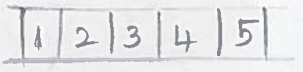






24/11/22

circular Queue.

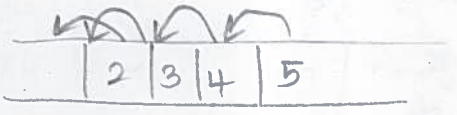


remove an element from Queue.

one space is free in queue but we cannot use those space

so one solution is

(i) replace all the elements present in the queue



↑ here you can insert new element.

But this process is costly (in terms of time and space consuming)

(ii) another solution is circular queue.

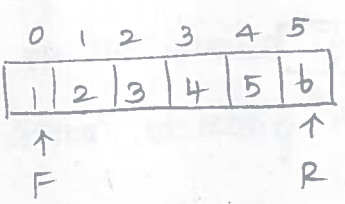
Insertion :- size - max element occupied by queue.

If front = 0 and rear = size-1, then the circular queue is full.

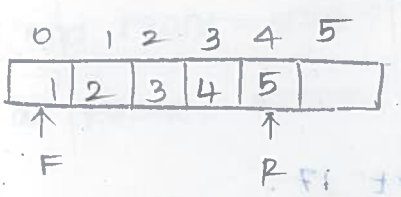
If rear != size-1, then rear will be incremented and the value will be inserted.

If Fron != 0 and rear != size-1, then it means that the queue is not full. So, set rear = 0 and insert the new element there.

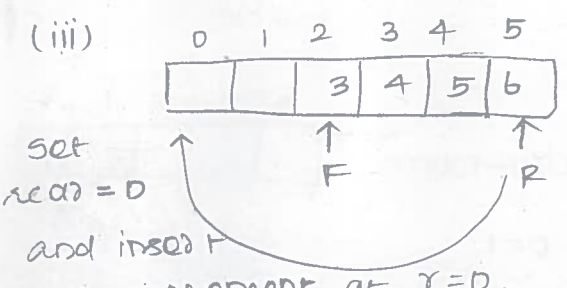
(i)



(ii)



(iii)



Algorithm :- (circular queue_insertion)

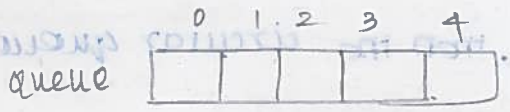
step 1 :- If (Front=0 and Rear = size-1) or Rear = Front-1
print "OVERFLOW"
GOTO step 4.

step 2 :- If Front = -1 and Rear = -1
set Front = rear = 0
Else if rear = size-1 and Front != 0
set Rear = 0
else
set rear = rear + 1.

step 3 :- set Queue [Rear] = Val.

step 4 :- Exit.

Insert 15, 17, 19, 20, Delete, 23, Delete, 25, 28



size = 5

F = R = -1

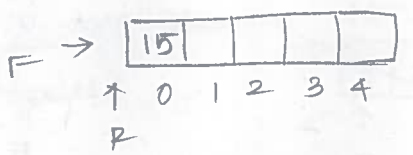
→ **Insert 15**

If F = 0 ✗

If F = -1 and R = -1 ✓

F = 0 R = 0

set queue[R] = queue[0] = 15



→ **Insert 17.**

If F = 0 ✓ and R = 5-1 ✗

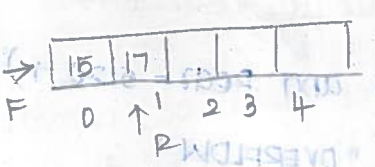
If F = -1 ✗

else if R = 5-1 ✗

else

R = R+1 = 0+1 ⇒ R = 1.

set queue[R] = queue[1] = 17



→ **Insert 19.**

If F = 0 ✓ and R = 5-1 ✗

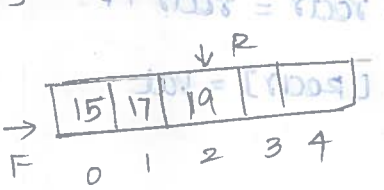
If F = -1 ✗

else if R = 5-1 ✗ ✗

else

R = R+1 = 1+1 ⇒ R = 2

set queue[R] = queue[2] = 19



continued in next page

Delete.
5, 28

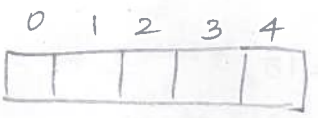
Deletion :-

If Front = -1 , then there are no elements in the queue. So an underflow condition will be reported.

If the queue is not empty and Front = rear then after deleting the element at the front, the queue becomes empty and so Front and Rear are set to -1.

If the queue is not empty and Front = size-1 then after deleting the element at Front, Front is set to 0.

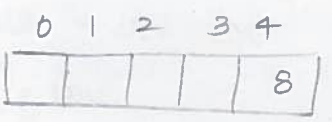
(i)



underflow

F=-1 R=-1.

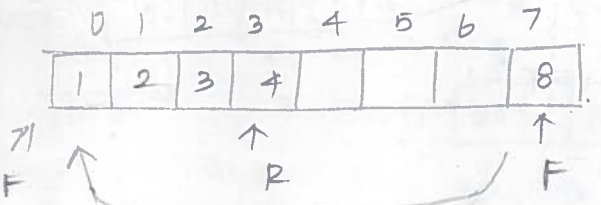
(ii)



F R

Delete the element & set F=R=-1.

(iii)



Delete this element and move F=0.

Algorithm :- (circularqueue_deletion)

Step 1 : If Front = -1
print "underflow"
goto step 4

Step 2 : set val = queue[Front]
print val

Step 3 : If Front = Rear
set Front = Rear = -1
else
if Front = size-1
set Front = 0
else
Front = Front

Step 4:
Exit

→ Insert 20

If F=0 ✓ and R=5-1 ✗

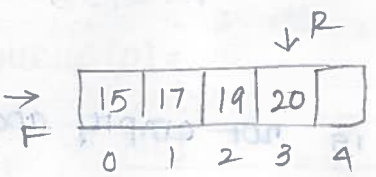
If F=-1 ✗

else if R= 5-1 ✗

else

R = R+1 = 2+1 ⇒ R=3

set queue[R] = queue[3] = 20



→ Delete

If F = -1 ✗

set val = queue[F] = queue[0]

val = 15 print (val = 15)

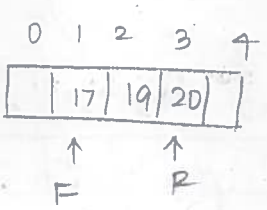
If F=R ✗

else

If F = 5-1 ✗

else

F = F+1 = 0+1 ⇒ F = 1



→ Insert 23

If F=0 ✗

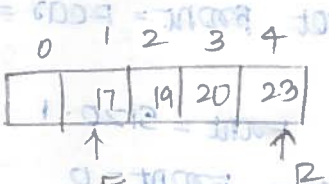
If F=-1 ✗

else if R=5-1 ✗

else

R = R+1 = 3+1 ⇒ R=4

set queue[R] = queue[4] = 23



→ delete

If F = -1 ✗

val = queue[F] = queue[1]

val = 17 print (val = 17)

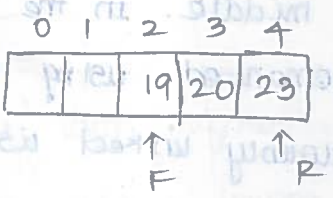
If F = R ✗

else

If F = 5-1 ✗

else

F = F+1 = 1+1 = 2 ⇒ F = 2



→ Insert 25

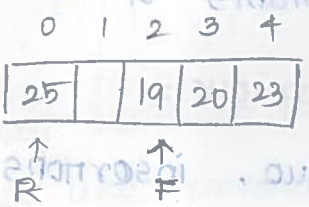
If F = 0 ✗

If F = -1 ✗

else if R = 5-1 = 4 ✓ and F != 0 ✓

set R = 0

else queue[R] = queue[0] = 25



→ Insert 28

If F = 0 ✗

If F = -1 ✗

else if R = 5-1 = 4 ✗

else

R = R+1 = 0+1 = 1 ⇒ R = 1

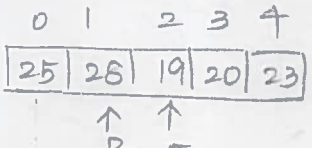
→ Insert 30

set queue[R] = queue[1] = 28

F = 0 ✗ → R = F-1

R = 2-1 = 1 ✓

print "OVERFLOW"



Dequeue : Double Ended queue

A dequeue is a list in which the elements can be inserted and deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either end or front.

However no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained LEFT and RIGHT, which point to either end of the deque.

The element in a deque extend from the LEFT end to the RIGHT end and since it is circular, Deque[N-1] is followed by Deque[0].

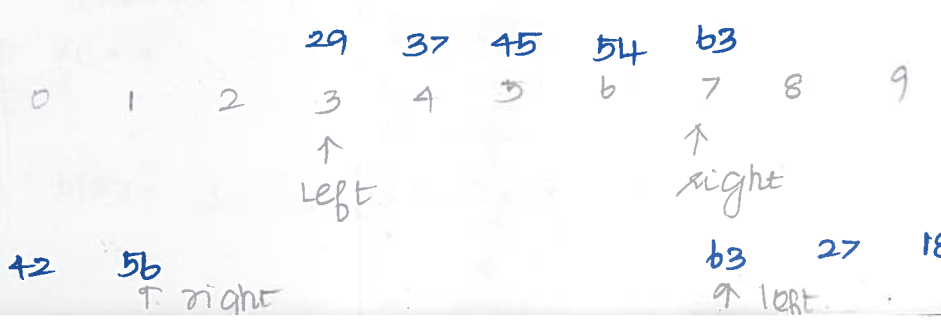
There are two variants of a double ended queue.

Input restricted dequeue :

In this deque, insertions can be done only at one of the ends, while deletions can be done from both ends.

Output restricted dequeue

In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.



Priority Queue:

→ A priority queue is a datastructure in which each element is assigned with a priority.

→ The priority of the element will be used to determine the order in which the elements will be processed.

→ The general rules of processing the elements of a priority queue are:

(1) An element with higher priority is processed before an element with a lower priority.

(2) Two elements with the same priority are processed on a first-come-first served (FCFS) basis.

→ Priority queues are widely used in operating systems to execute the highest priority process first.

Applications of Queues:

Queues are widely used as waiting lists for a single shared resources like printer, disk, CPU.

07/12/22

Co-3

Linked List :-

Memory Allocation :-

Static Allocation :-

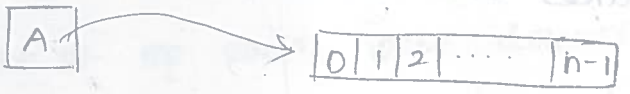
the memory that a data structure might possibly need is allocated all at once without regard for the actual amount needed at execution time.

Ex :- Arrays and strings.

Dynamic Allocation :-

Allocate memory for n integers → typecasting

```
int *A = malloc(n * sizeof(int));
```
(int *)



*(A) refers the

Accessing :-

```
if (A != NULL)
    for(i=0; i<n; i++)
        A[i] = 0
```
→ accessing using Array Format

(or)

```
for (i=0; i<n; i++)
    *(A+i) = 0;
```
→ accessing using pointer Format

points on memory allocation :

→ The memory allocator keeps track of what is allocated and what is free.

→ When memory is obtained from global pool, it contains garbage, and to properly initialize it.

→ Not to assume that successive requests will allocate contiguous memory blocks.

→ The failure of the request returns NULL value.

LINKED LIST :

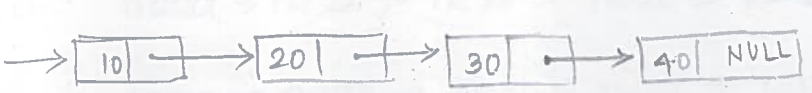
A linked list is a data structure which allows to store data dynamically and manage data efficiency.

Here is a pointer (called header) points the first.

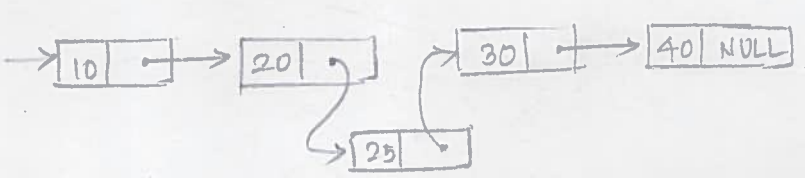
Successive nodes are connected by pointers.

Last element points to NULL.

It can grow or shrink in size during execution of programs.



25 insert



Advantages

→ Dynamic size

→ Ease of insertion / deletion.

Disadvantages

→ Random access is not allowed

→ Additional memory space requir for pointer.

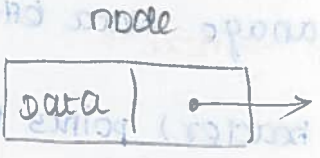
→ No spatial locality of referenc as elements are stored in non-contigous location.

Defining a Node of a Linked List :: (self referential lis

Each structure of the list is called a node, consists of two fields :

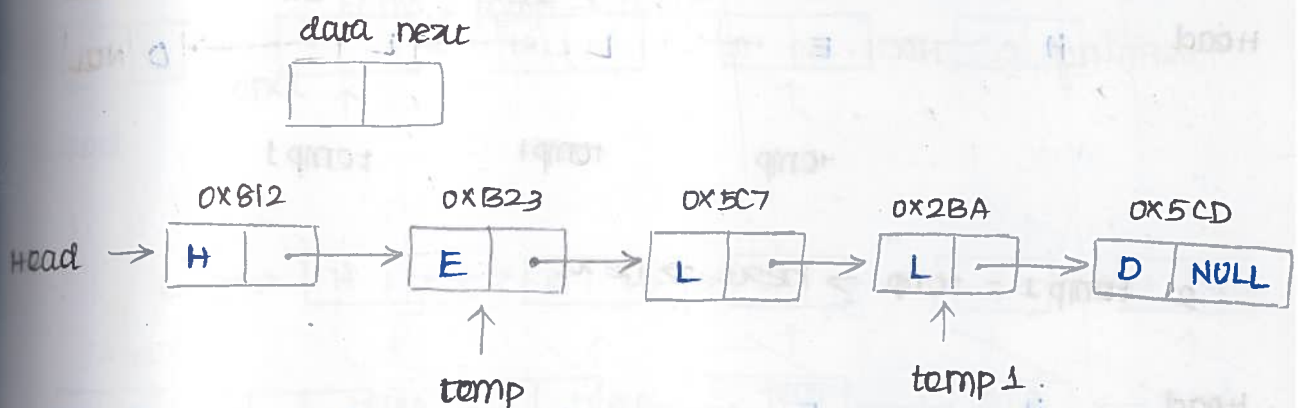
→ Item (or) Data

→ Address of the next item in the list (or) pointer to the next node in the list.




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

Creation of singly Linked List ::



Evaluate the Following Expressions:

(1) what address value is held by temp → next ?

0X5C7

(2) what does temp → next → data evaluate to ?

L

(3) what does (Head → next → next == temp) evaluate to ?

(False) 0X5C7 == 0XB23 X

Head → next == temp.

(4) using Head, temp. temp1 how can we access 'D' ?

Head : Head → next → next → next → next → data

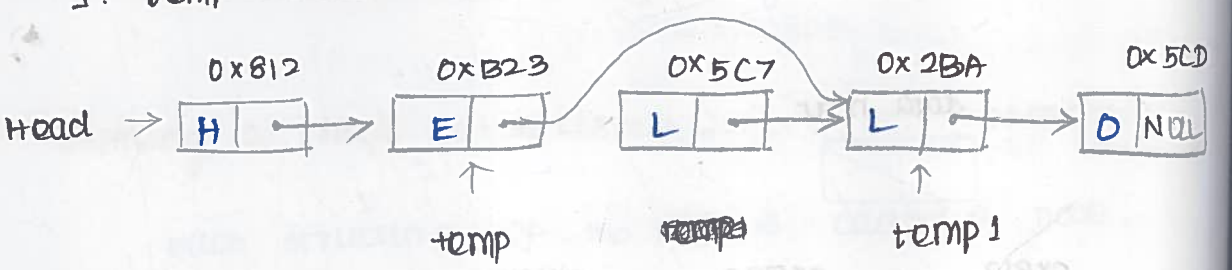
temp : temp → next → next → next → data

temp1 : temp1 → next → data

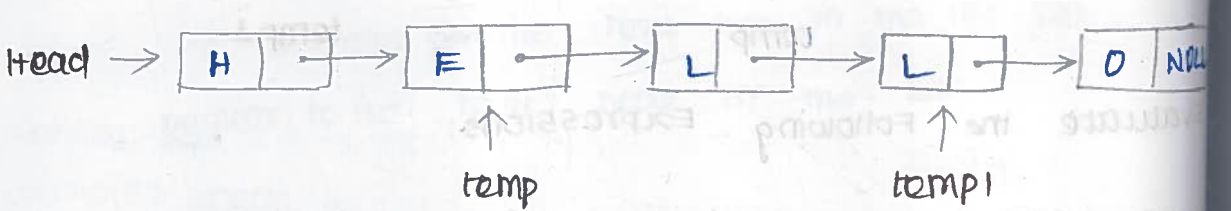
How does the following statement change our diagram.

1. temp → next = temp1
2. temp1 = temp → next → next
3. Head = head → next
4. temp1 = NULL.

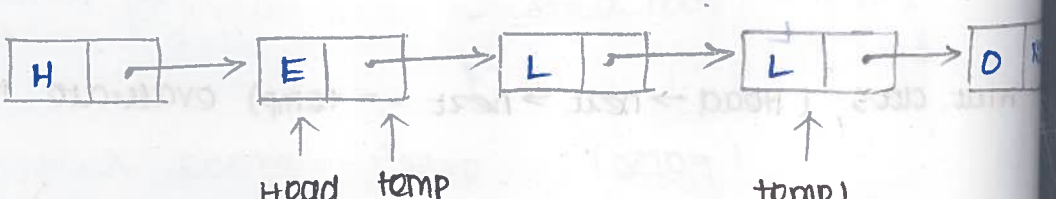
1. temp → next = temp1



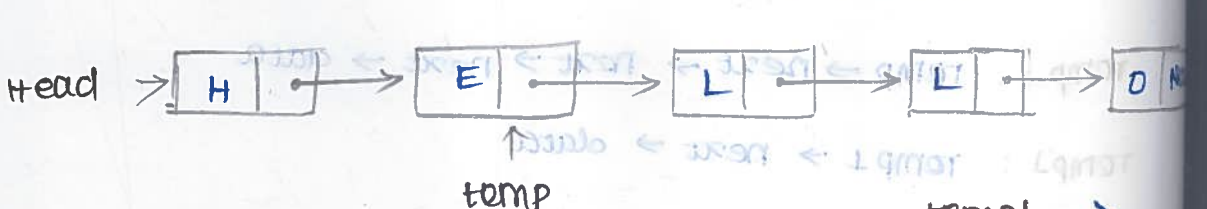
2. temp1 = temp → next → next.



3. Head = Head → next

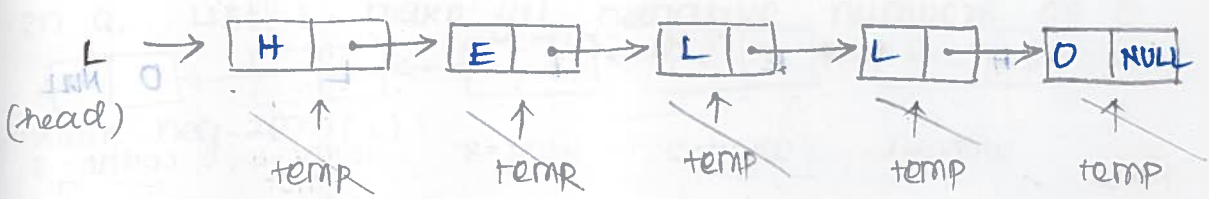


4. temp1 = NULL



Algorithm to print the values stores in a linked list :-

```
Algorithm print(L) :
  Begin
      temp = L
      while ( temp != NULL)
          begin
              print( temp → data)
              temp = temp → next
          end
  End
```



O/p :- HELLO .

16/12.

Algorithm to count the nodes in a linked list:

```
Algorithm Print(L):
        temp = L
        count = 0
        while( temp != NULL)
            count = count + 1
            temp = temp -> next
        print count
        end.
```

count = 1 count = 2 count = 3 count = 4 count = 5

Time complexity : O(n)

Write an algorithm that returns true if there is a node in the list with the value equal to x.

```
Algorithm search (L, x):
    temp = L
    while ( temp != NULL ) and ( temp -> data != x)
            { temp = temp -> next }
    if ( temp -> data == x)
            return true
    else
            return False
    end.
```

(or)

```
if ( temp == null)
        return False
else
        return true
```

Write an algorithm to print the value stored in the last node of the list.

```
Algorithm print_lastnode (L):
        temp = L
        if temp == L :
            print Empty List
            return
        while ( temp -> next != null)
            { temp = temp -> next }
        print temp -> data "as the last node data".
        return
End.
```

print statement.

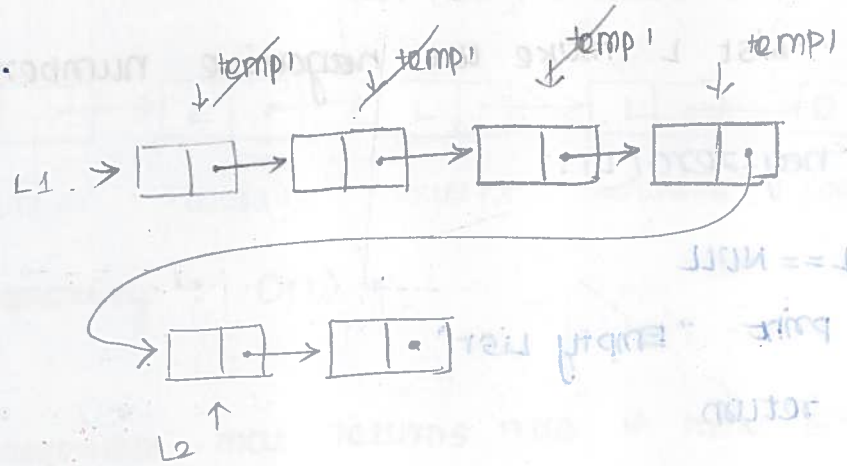
Given a list L make all negative numbers as 0.

```
Algorithm neg-zero(L):
    if L == NULL
        print "Empty List"
        return
    else
        temp = L
        while ( temp != NULL)
        {
            if temp -> data <= 0 :
                temp -> data = 0
            temp = temp -> next
        }
    return
end.
```

Given two List named L1 and L2, have to append L2 at the end of L1.

Algorithm append (L1, L2):

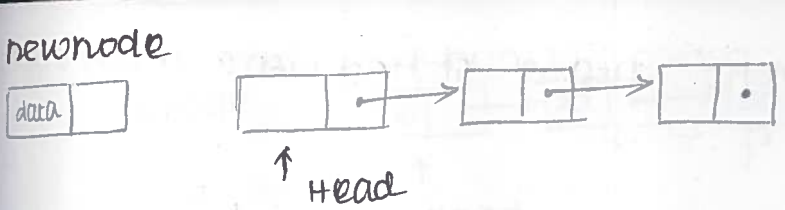
```
if L1 == NULL && L2 == NULL
    return
else if L1 == NULL:
    return L2
else if L2 == NULL:
    return L1
else:
    temp1 = L1
    while ( temp1 -> next != NULL)
    { temp1 = temp1 -> next }
    temp1 -> next = L2
    return L1
end.
```

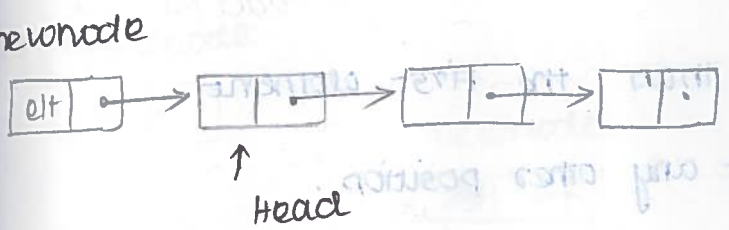


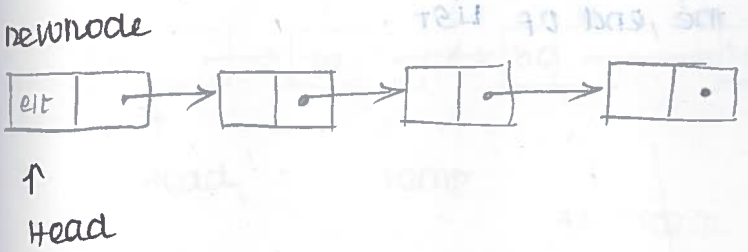
Insertion into a Linked List :

Algorithm InsertAtHead (Head, elt)

```
Begin
    newnode = getnode (NODE)
    newnode -> data = elt
    newnode -> next = Head
    Head = newnode
End
```







Algorithm InsertAtEnd (Head, elt)

Begin

newnode = getnode (NODE)

newnode → data = elt

if (Head == NULL) :

{ newnode → next = NULL

Head = newnode

return }

else {

temp = Head

while (temp → next != NULL)

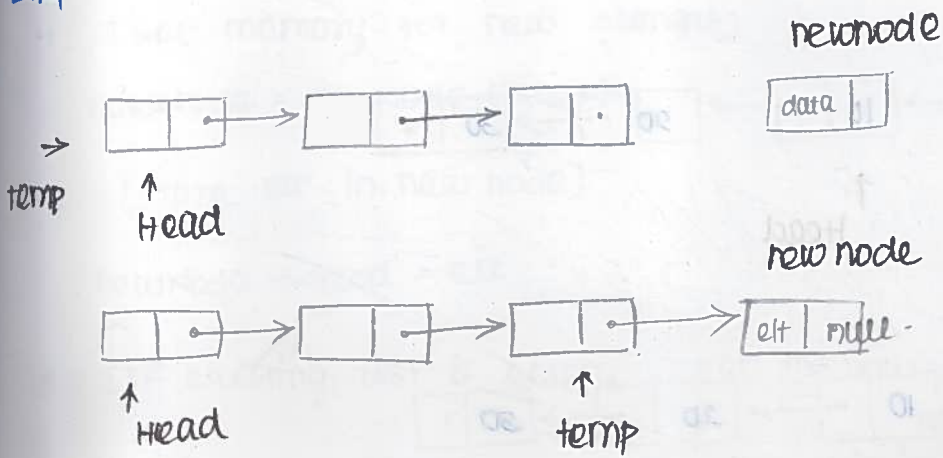
{ temp = temp → next }

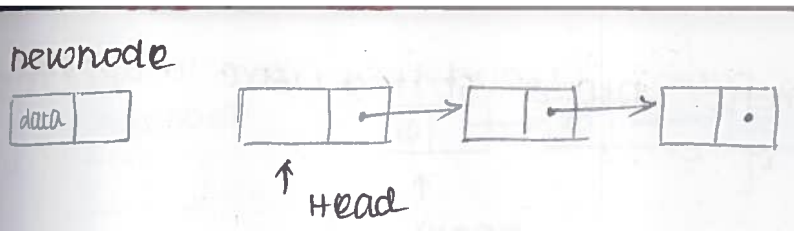
temp → next = newnode

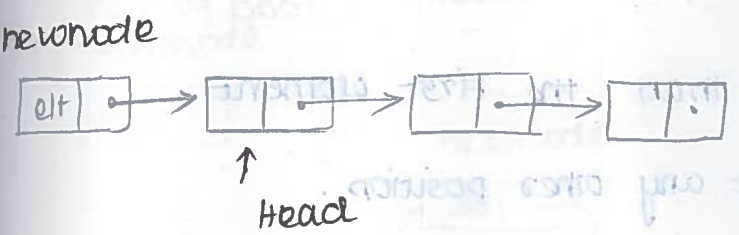
newnode → next = null . }

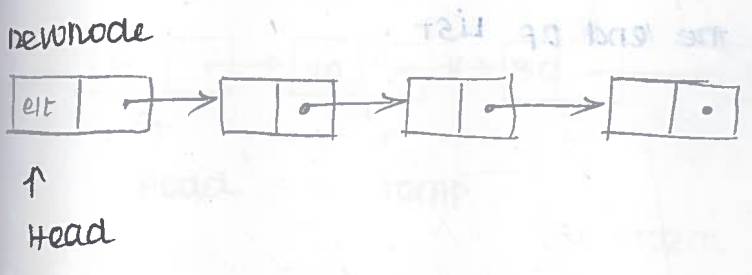
return

End









Algorithm InsertAtEnd (Head, elt)

Begin

newnode = getnode (NODE)

newnode → data = elt

if (Head == NULL) :

{ newnode → next = NULL

Head = newnode

return }

else {

temp = Head

while (temp → next != NULL)

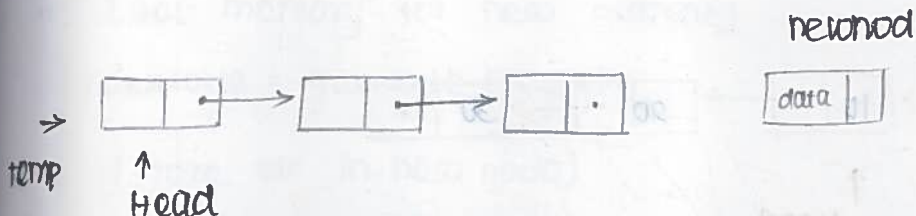
{ temp = temp → next }

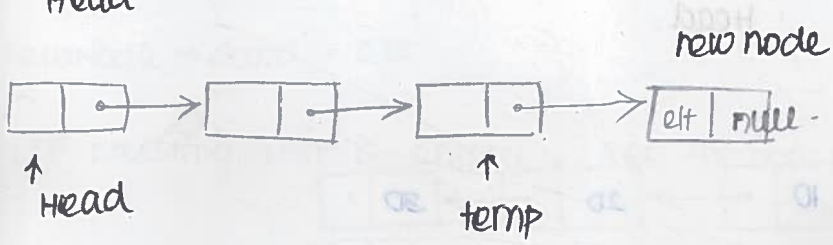
temp → next = newnode

newnode → next = null . }

return

End





Insert an element into a sorted List :

case :-

1. Empty List → head pointer changes.
2. Element less than the First element
3. Element at any other position.
4. Adding at the end of List.

case 1 :-

Head = Null Insert 25

newnode

newnode

newnode

Head.

case 2 :-

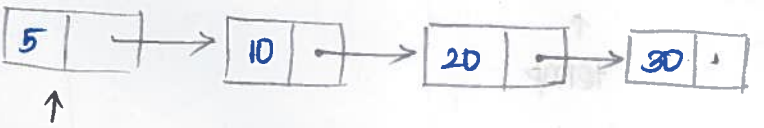
newnode

Head

newnode

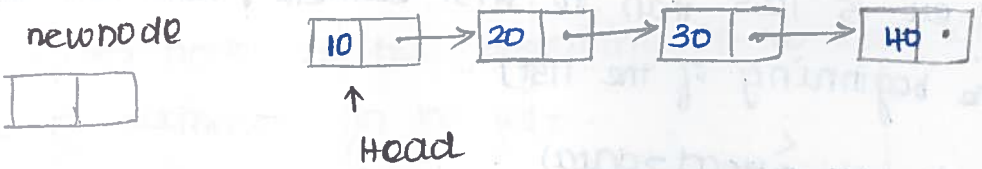
Head

newnode

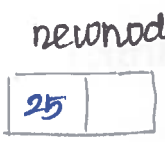


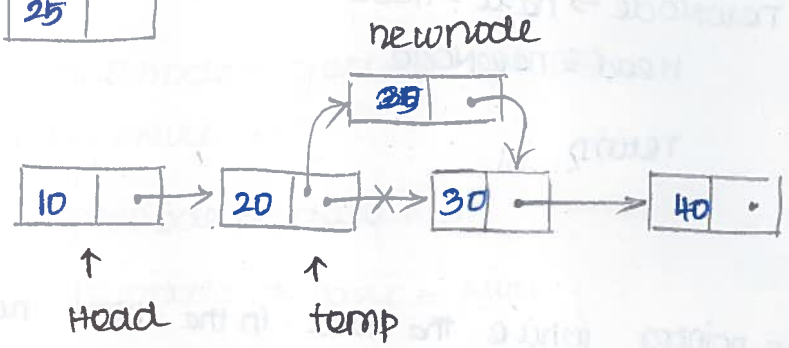
Head

se 3 :



pointer changes.





gorithm InsertSortedList (Head, elt) :

1. Get memory for new element
2. store elt in new node
3. If existing list is empty, set the new node as resultant list
4. If elt is less than the first element, add new node in the beginning of the list.
5. Move the pointer while the data in the next node is smaller than the elt and insert the new node

gorithm Insert sorted List (Head, elt) :

1. [Get memory for new element]
   newNode = getNode (NODE)
2. [store elt in new node]
   newNode → data = elt
3. [If existing list is empty, set the new node as resultant list]

```
if (Head == NULL)
{   newNode → next = NULL
    Head = newNode
    return  }
```

4. [If elt is less than the first element, add newNode in the beginning of the list]

```
if (elt < Head->data)
{
    newNode->next = Head
    Head = newNode
    return
}
```

5. [Move the pointer while the data in the next node is smaller than the elt and insert the new node]

```
temp = head
```

6. While (temp->next->data < elt)

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

7. newNode->next = temp->next

8. temp->next = newNode

9. Return

16/12/22

1. Add a header node at the beginning of the list with the count of elements in the list.

Algorithm count_node_add_begin (L):

Begin newnode = getnode (NODE)

if L==NULL : {

newnode → data = 0

newnode → next = NULL

L = newnode

return L }

else : {

temp = L

count = 0

while temp != NULL :

{ count ++

temp = temp → next }

newnode → data = count

newnode → next = L

L = newnode

return L }

End

2. Add a node at the End of the list and store the no of negative elements in the newnode.

Algorithm neg_value_add_end (L):

Begin

newnode = getnode (NODE)

if L==NULL : {

newnode → data = 0

newnode → next = NULL

L = newnode return L }

```
else : {
    temp = L
    neg_count = 0
    while temp -> next != null : {
        if temp -> data < 0 : {
            neg_count ++
            temp = temp -> next
        } }
    if temp -> data < 0 :
        neg_count ++
    newnode -> data = neg_count
    temp -> next = newnode
    newnode -> next = NULL
    return L }
End.
```

4. Add a value to all the elements of the list.

```
Algorithm  Addvalue (L, val)
Begin
    if L == NULL : {
        print Empty List
        return L
    }
    else : {
        temp = L
        while temp != NULL : {
            temp -> data = temp -> data + val
            temp = temp -> next
        }
        return L
    }
End
```

3. write an algorithm to merge two sorted linked list.

```
Algorithm merge_sorted_list( L1, L2)
Begin
    L3 = getnode (NODE)
    temp1 = L1
    temp2 = L2
    temp3 = L3
  while temp1 != NULL and temp2 != NULL :
    begin
        if temp1 -> data <= temp2 -> data {
            temp3 -> next = temp1
            temp1 = temp1 -> next }
        else {
            temp3 -> next = temp2
            temp2 = temp2 -> next }
        temp3 = temp3 -> next
    End
  if temp1 == null :
        temp3 -> next = temp2
  else if temp2 == null:
        temp3 -> next = temp1.

  return L3
End.
```

For an alternate solution given by Mam, see after two pages.

5. Addition of numbers represented by lists.

Merging of two sorted lists:

Input

Head1 → 4 → 8 → 15 → 19 → NULL

Head2 → 7 → 9 → 10 → 16 → NULL

Output

Head1 → 4 → 7 → 8 → 9 → 10 → 15 → 16 → 19 → NULL

Algorithm mergeList (List1, List2):

```
Begin
    if List1 == NULL
        return List2
    else if List2 == NULL
        return List1

    if (List1 -> data <= List2 -> data)
    {
        mergedList = List1
        List1 = List1 -> next
    } else {
        mergedList = List2
        List2 = List2 -> next
    }
    mergedTail = mergedList
    while (List1 != NULL AND List2 != NULL) {
        temp = NULL
        if (List1 -> data <= List2 -> data) {
            temp = List1
            List1 = List1 -> next
        }
```

```
        else {
            temp = List2
            List2 = List2 -> next
        }
        mergedTail -> next = temp
        mergedTail = temp
    }
    if (List1 != NULL)
        mergedTail -> next = List1
    else if List2 != NULL
        mergedTail -> next = List2

    return mergedList

End.
```

Write an algorithm, to return the length of their long linked list among the two lists.

```
Algorithm longList (List1, List2)
Begin
    if List1 == NULL and List2 == NULL
        "print Both the lists are Empty"
        return
    else if List1 == NULL and List2 != NULL
        return List2
    else if List1 != NULL and List2 == NULL
        return List1
```

```
temp1 = List 1

temp 2 = List 2

count 1 = 0,  count 2 = 0

while temp1 != NULL
{ count 1 += 1
  temp1 = temp1 -> next
}

while temp2 != NULL
{
   count 2 += 1
   temp 2 = temp2 -> next
}

if count 1 == count 2
      print " Both the lists are of same size "

      return

else if count 1 < count 2

      return List 2

   else if
      return List 1


End .
```

without using two while loops it can be done within the single while loop also.

```
temp1 = List1
temp2 = List2
while temp1 != NULL and temp2 != NULL
{
    temp1 = temp1 -> next
    temp2 = temp2 -> next
}
if temp1 == NULL and temp2 == NULL:
        print "Both the lists are of same size"
else if temp1 != NULL:
        return List1
else if temp2 != NULL:
        return List2
```

ize"

End.

Delete a node :.

1. Delete the first node in the list
2. Delete the last node
3. Delete the node that has the given key
4. Delete the list.

Delete the first node in the list

Algorithm DeleteFirst (List)

Begin

if List = = NULL

return

temp = List

List = List → next

Free (temp)

return

End.

Delete the last node.

Input

List 1

Output

List 2

Algorithm DeleteLast (List)

Begin

if (List = = NULL)

return NULL

if (List → next = = NULL)

free (List)

return NULL

temp = List

while (temp → next → next != NULL)

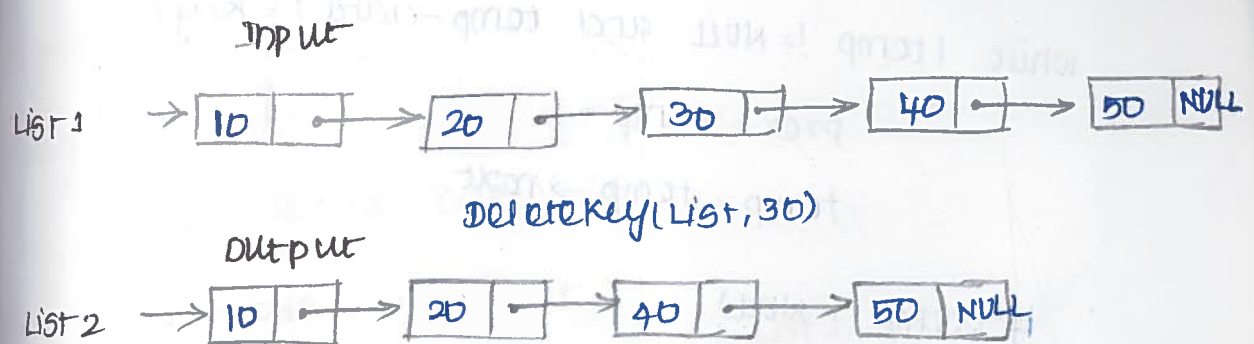
temp = temp → next

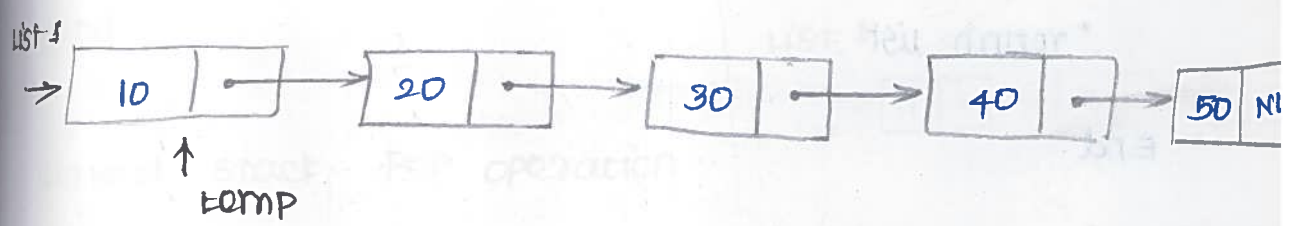
free (temp → next)

temp → next = NULL return List

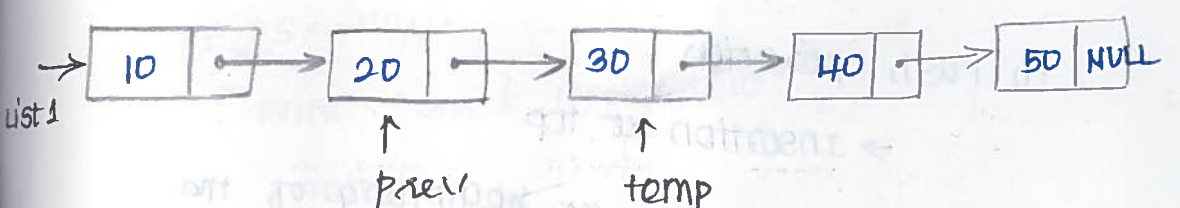
End

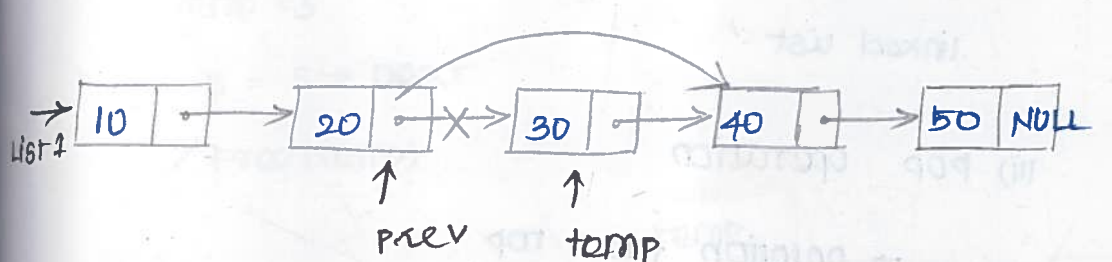
Delete the node with the given key.





Prev = NULL





Algorithm DeleteKey (List, key):

Begin

if (List == NULL)

return NULL

Prev = NULL

if List → data == key

temp = List

List = List → next free(temp)

```
temp = list
while (temp != NULL and temp->data != key)
        prev = temp
        temp = temp -> next
if (temp != NULL)
        prev -> next = temp -> next
        free(temp)
return list.
End.
```

Linked stack :-

Stacks implemented using linked lists.

(i) Push operation

→ Insertion at top

→ Insertion at beginning of the linked list

(ii) Pop operation

→ Deletion from top

→ Deletion of first node from the linked list.

Linked stack - Push Operation :

Algorithm PUSH (S, elt) :

Begin

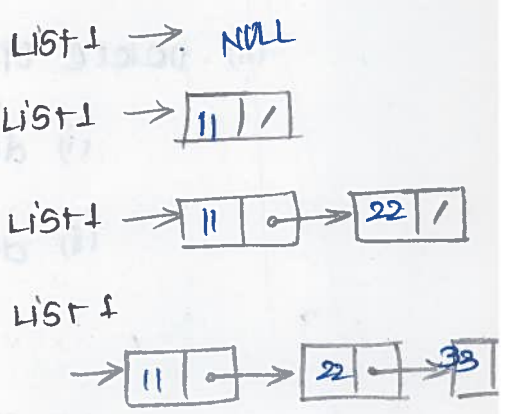
newNode = getnode(NODE)

newNode → data = elt

newNode → next = S

S = newNode

End



Linked stack - POP operation :

Algorithm POP (S, elt) :

Begin

If (S == NULL)

print "stack underflow"

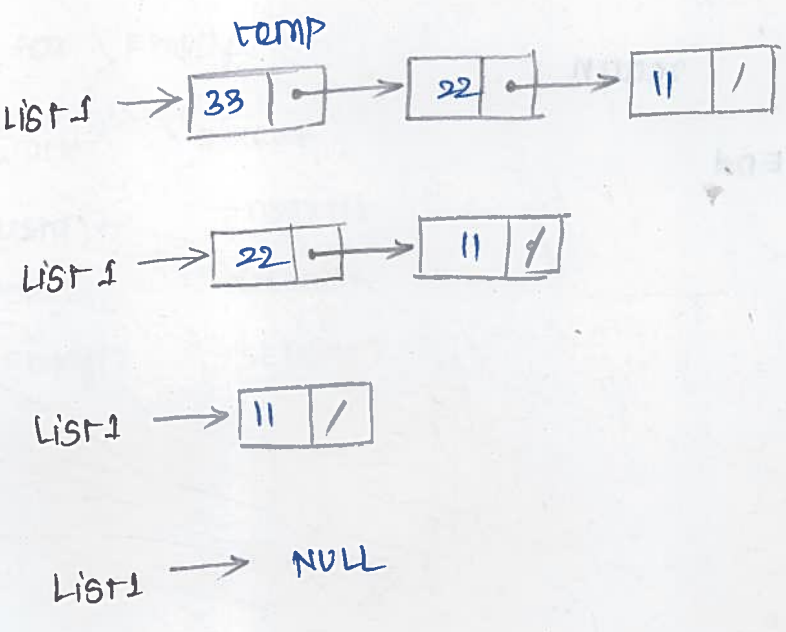
return

temp = S

S = S → next

Free (temp)

End.



Linked Queue :-

(i) Insert operation

(i) Insert at rear end

(ii) insert at the tail of list

(ii) delete operation

(i) deletion from the front

(ii) delete from the head of the list.

Algorithm LQInsert(Q, elt) :

Front & Rear pointers NULL

Begin

```
newNode = getNode(Node)
newNode -> data = elt
newNode -> next = NULL
If Rear == Front == NULL
        Rear = Front = newNode
else {
        Rear -> next = newNode
        Rear = newNode
    }
return
```

End

Algorithm LLDelete (Q)

Begin

If (Rear == Front == NULL)

{ print " Q is Empty"

return }

If (Front == Rear)

{ free (Rear)

Front = Rear = NULL

return }

temp = Front

Front = Front → next

free(temp)

return

End

Exam :

1. pointers needed
2. Elements pointers pointing.
3. Operations that can be performed. → Algorithm.
4. conditions for Empty.
5.

Stack	queue
push()	insert()
pop()	delete()
isEmpty()	isEmpty()

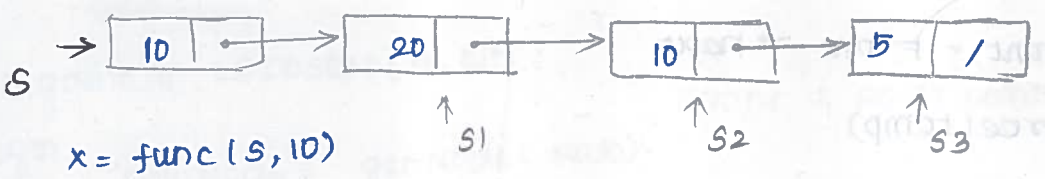
What does the function do?

counts the number of nodes with value c.

```
Algorithm func(S,C)
    if (S==NULL)
        return 0
    x = func(S->next, C)
    if (S.data==C)
        x++
    return x
```

It counts the no of occurrences of c.

c=10



x = func(S, 10)

10==10
x++
return x=1

x = func(S1, 10)

x=1
10==10
x++
x=1

x = func(S2, 10)

return x=0.

x = func(S3, 10)

x = func(S4, 10)

S4==NULL
return 0.

Final result :- 2.

26/12/22

Reverse a singly linked list :-

Algorithm reverseList (Head) :

Begin

last = NULL

temp = Head

while (temp != NULL)

{

nextNode = temp -> next

temp -> next = last

last = temp

temp = nextNode

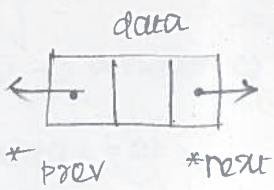
}

Head = last

End

Doubly Linked List :

Linked list with two pointers pointing to previous and next element of the list.




```
struct DLNODE{
    struct DLNODE * prev;
    int data;
    struct DLNODE *next;
}
```

Insertion :

Insert a newNode after the Element

```
Algorithm insertDLL ( Head, elt, value)
Begin
    if (Head == NULL)
        print " Empty list"
        return
    temp = head
    while ( temp -> data != elt AND temp != NULL)
      { temp = temp -> next }
    if ( temp != NULL)
      { newNode = getnode (NODE)
        newNode -> data = value
        newNode -> next = temp -> next
        newNode -> prev = temp
        if ( temp -> next != NULL)
            temp -> next -> prev = newNode
        temp -> next = newNode }
    else
      { print " Error, invalid element" }
    return Head ;
End
```

Deletion in DLL:

Algorithm DeleteNodeDLL (Head, P):

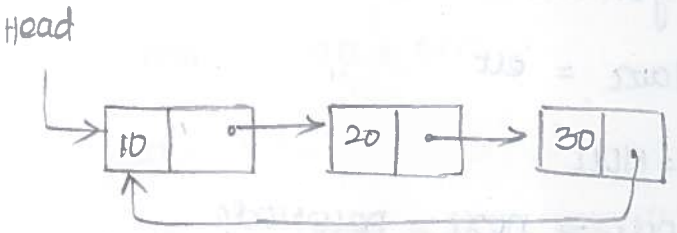
Begin

```
If Head ==NULL
    print "Empty List"
    return
If P == Head
{   Head = Head->next
    Head -> prev = NULL
     free(P)
     return Head
}
P -> prev -> next = P -> next
P -> next -> prev = P -> prev
free(P)
return Head
```

End

circular Linked List (CLL)

The next field of last node points to the first node can traverse the list from any node.



count the number of nodes in a CLL:

```
Algorithm countCLL(Head)
Begin
    temp = Head
     count = 0
    if (head != NULL) {
            do {
                temp = temp->next
                 count = count +1
            } while (temp != Head)
    }
    return count
End.
```

Insertion at the beginning of CLL:

Algorithm InsertBegCLL (Head, elt):

```
Begin
    newNode = getnode(NODE)
    newNode->data = elt
    if Head == NULL :
        newNode -> next = newNode
        Head = newNode
        return Head
    temp = Head
    newNode -> next = Head
    while (temp->next != Head)
        temp = temp->next
    temp -> next = newNode
    Head = newNode
End
```

old Link.

Insertion at end of CLL :-

```
Algorithm InsertEndCLL(Head, elt) :
Begin
        newNode = getNode(NODE)
        newNode → data = elt
        if Head == NULL :
                newNode → next = newNode
                Head = newNode
                return Head
        temp = Head
        while temp → next != Head :
                temp = temp → next
        temp → next = newNode
        newNode → next = Head
        return Head
End.
```

(1)
Delete First node of the circular singly linked list.

(2)
Delete Last node of the circular singly linked list.

28/12/22

CO4 Hashing:

Many application deals with lots of data

the look ups are time critical.

search

Array : O(n)

Binary search : O(log n)

↳ constant number of comparison

Better Data structure : O(1)

↳ independent on size of n.

※

Hashing:

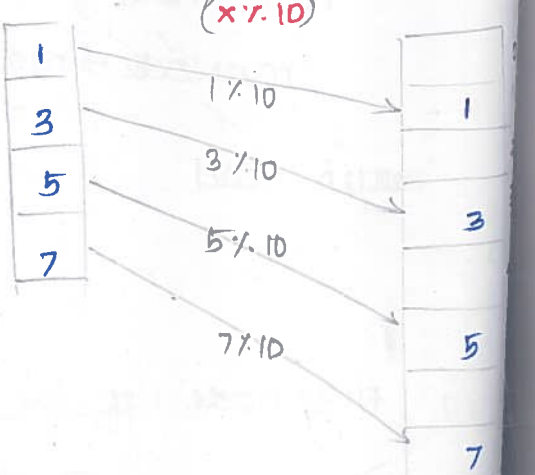
Mapping key to address

h(key) = address

hash function.

Hash FUNCTION :-

(x % 10)



Finding a Hash Function:

Assume that N = 5,

values need to be inserted are : cab, bea. bad.

Let a=0, b=1, c=2, etc...,

Define H such that

H [data] = (Σ characters) mod N

H [cab] = (2+0+1) mod 5 = 3

H [bad] = (1+0+3) mod 5 = 4

irrespecti

position of ch

all gives s

mod

good hash function must

Be easy to compute

Avoid collisions

↳ different keys mapping to the same memory address by applying the hash function.

Methods of Hashing :-

Division Method :-

The hash function divides the value of k by M and then uses the Formula:

$$h(k) = k \bmod M$$

↳ better to have it as prime

here k = key.

M = table size

k = 12345

M = 95

h(12345) = 12345 % 95

= 90

k = 1276

M = 11

h(1276) = 1276 % 11

= 0.

Mid square Method :

It computes two steps to compute the hash value -

1. square the value of key (i.e k * k)
2. Extract the middle r digits as the hash value.

Formula :

$$h(k) = k * k$$

hash table has table 100 memory locations.

r = 2 because two digits are required to map the key to memory location.

k = 60 k x k = 60 x 60 = 3600

h(60) = 60 hash value = 60

digit Folding Method :

2steps :

1. divide the key value k into a no of parts i.e.,

$k_1, k_2, \cdots, k_n$

where each part has the same no. of digits except for the last part that can have lesser digits than the other parts.

2. Add individual parts

3. the hash value is obtained by ignoring the last

$k = 12345$ two bit address

$k_1 = 12 \quad k_2 = 34 \quad k_3 = 5$

$s = k_1 + k_2 + k_3$

$= 12 + 34 + 5$

$= 51$

$h(k) = 51$

$k = 76951$ two bit address.

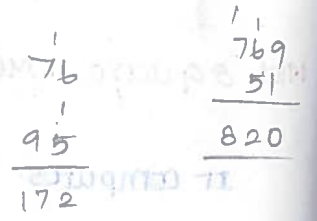
$k_1 = 76 \quad k_2 = 95 \quad k_3 = 1$

$s = k_1 + k_2 + k_3$

$= 76 + 95 + 1$

$= 172$

$h(k) = 72$



three bit address

$k_1 = 769 \quad k_2 = 51$

$s = k_1 + k_2$

$= 769 + 51$

$= 820$

$h(k) = 820$

Multiplication Method :

1. choose a constant value A such that $0 < A < 1$.
2. Multiply key value with A.
3. Extract fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M
5. The resulting hash value is obtained by taking the floor of the result obtained.

$k = 12345$

$A = 0.357840$

$M = 100$

$h(12345) = \text{floor}\ [100\ (12345 * 0.357840)]$

$= \text{floor}\ [100\ (4417.5348)]$ → taking only decimal portion of the values.

$= \text{floor}\ [100\ (0.5348)]$

$= \text{floor}\ (53.48)$

$= 53$

properties of good hash function :

⇒ must return number 0, ..., tablesize.

⇒ should be efficiently computable - O(1) time.

⇒ should not waste space unnecessarily.

* For every index there is atleast one key that hashes to it.

* Load factor lambda λ = (no. of. keys / table size)

⇒ should minimize collisions

= different keys hashing to same index.

collisions :

→ try inserting "abc", "cba", "bca"

[a=0, b=1, c=2]

H[abc] = 3

H[cba] = 3

H[bca] = 3

→ they all map to the same location

→ this is called "collision"

→ collisions can be reduced with a selection of a good hash function.

collision resolution methods :

separate chaining

linear probing

double hashing

quadratic probing.

separate chaining

/open hashing - collisions are stored outside the table

closed hashing/ open addressing - collisions result in storing

one of the records at another slot in the table.

Linear Probing:

Table remains a simple array of size N.

On insert(x), compute f(x) mod N.

If the cell is full, find another by sequentially searching for the next available slot (f(x)+1, f(x)+2, etc.)

Linear probing function can be given by

$$h(x,i) = (f(x)+i) \bmod N \quad (i=1,2,\dots)$$

Hash function:

H(k) = k mod 10

hash(89,10) = 9

hash(18,10) = 8

hash(49,10) = 9 (collision)

0th location is filled.

hash(58,10) = 8 (collision)

1st location is filled

hash(9,10) = 9 (collision)

2nd location is filled

hash(55,10) = 5

49	0
58	1
9	2
	3
	4
55	5
	6
	7
18	8
89	9

consider H(key) = key Mod 6 (assume N=6)

H(11) = 5

H(10) = 4

H(17) = 5 (collision)

H(16) = 4 (collision)

H(23) = 5 (collision)

0			17	17	17
1				16	16
2					23
3					
4		10	10	10	10
5	11	11	11	11	11

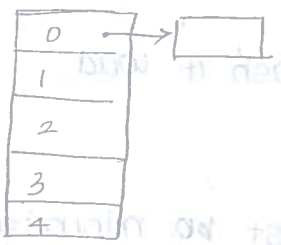
Linear probing- Deletion.

Item in a hash table connects to other

8/12/22

separate chaining :-

collisions can be resolved by solving by creating a list of keys that map to the same value.

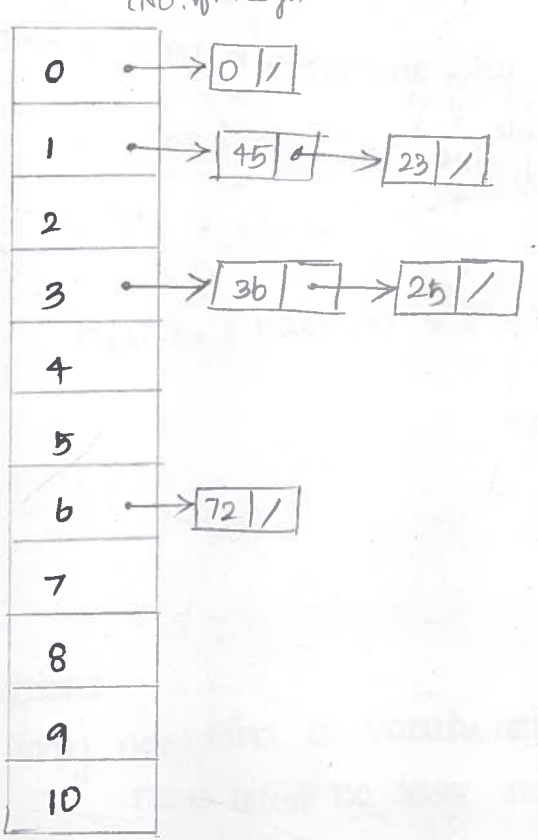


use an Array of N linked lists where N is the table size.

25, 23, 72, 45, 36, 44 use the Hash function $H(K) = K \% 11$ construct the hash table use separate chaining to resolve collission.

(table size)

M = 11 N = 6 (No. of. keys)



$h(25) = 25 \% 11 = 3$ ✓

$h(23) = 23 \% 11 = 1$ ✓

$h(72) = 72 \% 11 = 6$ ✓

$h(45) = 45 \% 11 = 1$ ✓

$h(36) = 36 \% 11 = 3$ ✓

$h(44) = 44 \% 11 = 0$ ✓

collision resolved by using insertion at beginning of the linked list.

(i) no of probes required to find an element.

↓

no of comparisons made.

worst case :- O(n)

when all the elements mapped to the same addressing

Load factor (Open Addressing)

Load factor λ of a probing hash table is the fraction of the table that is full.

Load factor ranges from 0 (empty) to 1 (completely

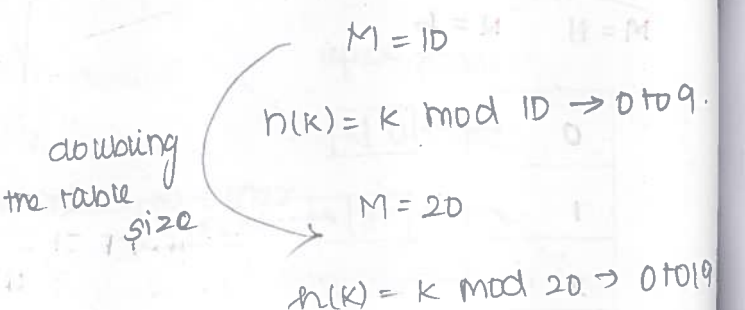
Better to keep the load factor under 0.7

Double the table size and rehash if load factor gets high.

cost of Hash function f(x) must be minimized.

When collisions occur, linear probing can always find an empty cell.

But clustering can be a problem.



Drawbacks of linear probing.

↳ primary clustering.

Quadratic Probing. - Avoid primary clustering to some extent.

A form of closed Hashing - open addressing.

spread out the search for an empty slot.

increment by i^2 instead of i

↑ quadratic probing

↑ linear probing.

$h_i(x) = (\text{Hash}(x) + i^2) \% \text{tablesize}$.

resolve collision by examining certain cells $(1, 4, 9, \ldots)$ away from the original probe point. $(1^2, 2^2, 3^2, 4^2, \ldots)$

Limitations :

May not find a vacant cell;

Table must be less than half full.

May leads to the cycle.

25, 33, 72, 45, 36, 44.

Hash Function : $H(K) = K \% 11$

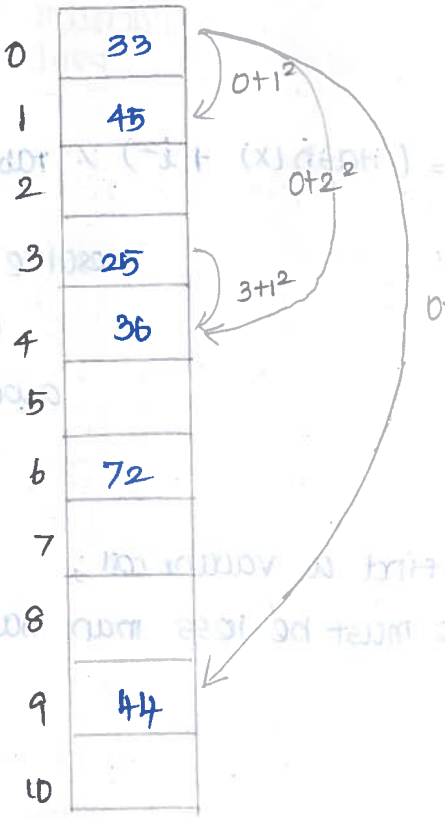
Key	25	33	72	45	36	4
Hash value $h(K)$	3	0	6	1	3 ↑ collision	

0	33
1	45
2	44 ←
3	25
4	36 ←
5	
6	72
7	
8	
9	
10	

Primary clustering.

Linear probing.

Quadratic Probing



Quadratic probing.

Pseudo-random probing.

An alternative to quadratic probing

Idea: generate a list of random numbers and use that list to determine the next index.

E.g., {0, 8, 3, 4, 7, 2, 9, 6, 1, 5}

index = h(k) + randomarr[0]

If collision, index = h(k) + randomarr[1]

then

h(k) + randomarr[2], etc.,

very quick to calculate!

maybe a bit better than quadratic probing.

Double Hashing. closed Hashing.

Idea: spread out the search for an empty slot by using a second hash function.

NO primary or secondary clustering.

$h_i(x) = (Hash_1(x) + i * Hash_2(x)) \bmod tablesize$

for $i = 0, 1, 2, \ldots$

Good choice of $Hash_2(x)$ can guarantee not getting "stuck"

14, 8, 21, 2, 7

$Hash_1(x) = x \bmod 7$

$Hash_2(x) = 5 - (x \bmod 5)$

0	14
1	8
2	2
3	7
4	21
5	
6	

$14 \Rightarrow 14 \bmod 7 = 0$ ✓

$8 \Rightarrow 8 \bmod 7 = 1$ ✓

$21 \Rightarrow 21 \bmod 7 = 0 \rightarrow$ collission

$5 - (21 \bmod 5)$

$5 - 1 = 4$ ✓

$2 \Rightarrow 2 \bmod 7 = 2$ ✓

$7 \Rightarrow 7 \bmod 7 = 0 \rightarrow$ collission.

$5 - (7 \bmod 5)$

$= 5 - 2 = 3$

probes :

14	8	21	2	7
1	1	2	1	2

probe sequence :

0^{th} probe $= h(k) \bmod M$

1^{st} probe $= (h(k) + 1(g(k))) \bmod M$

2^{nd} probe $= (h(k) + 2 * g(k)) \bmod M$

3^{rd} probe $= (h(k) + 3 * g(k)) \bmod M$

and so on.

Rehashing :-

Idea :- When the table gets too full, create a bigger table (usually 2x larger) and hash all the items from the original table into the new table.

When to rehash ?

half full ($\lambda = 0.5$)

when an insertion fails.

Cost of rehashing ?

Methods to improve hashing performance.

use good hash function

user larger table size

use good collision resolution methods.

LINKED LISTS :

Multiply linked list :-

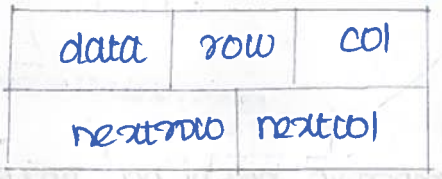
Node has multiple pointer fields.

sparse matrix representation.

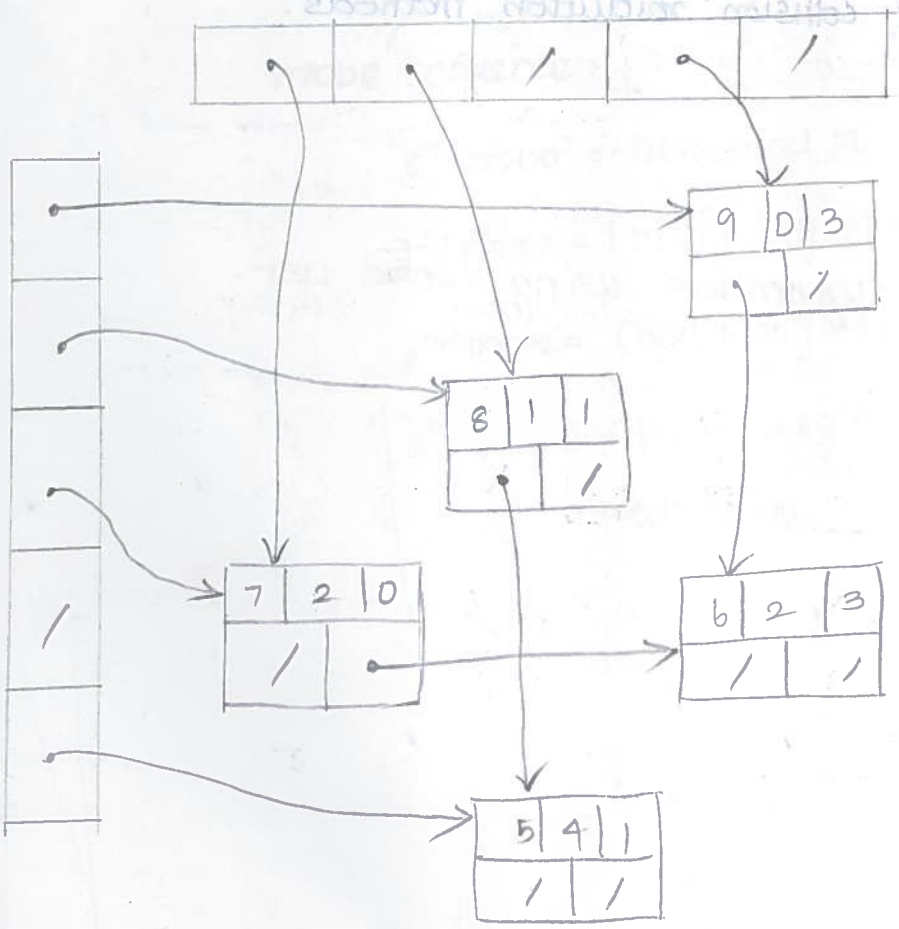
Every node in the linked list has links to several nodes of the list.

0	0	0	9	0
0	8	0	0	0
7	0	0	6	0
0	0	0	0	0
0	5	0	0	0

Node



sparse matrix representation using multiply linked list.

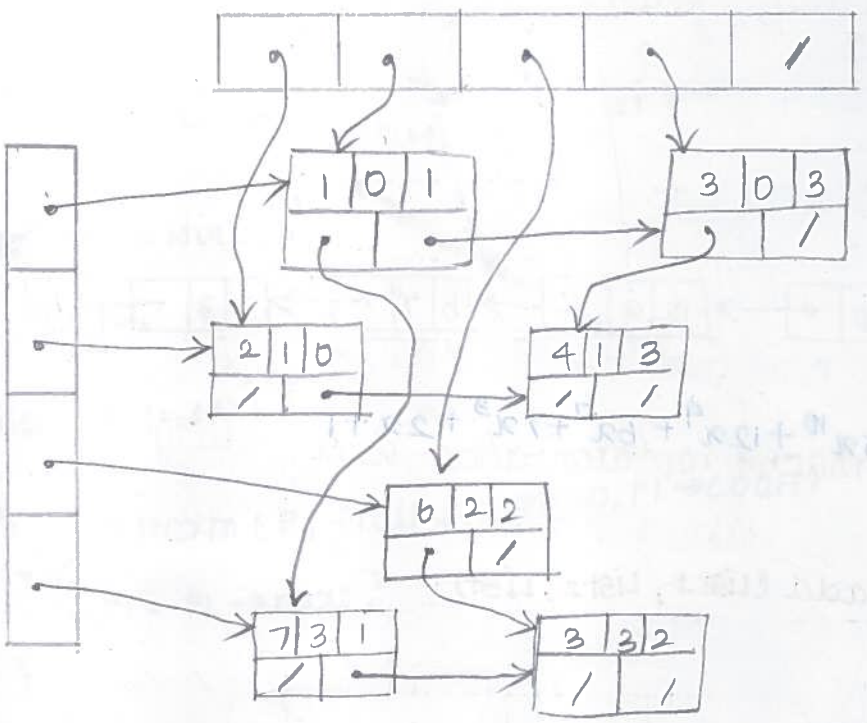


0 1 0 3 0

2 0 0 4 0

0 0 6 0 0

0 7 3 0 0

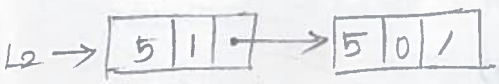


representation of polynomials using linked list.

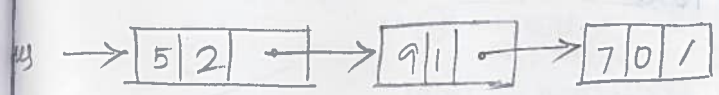
Node structure

$5x^2 + 4x + 2$

$5x + 5$



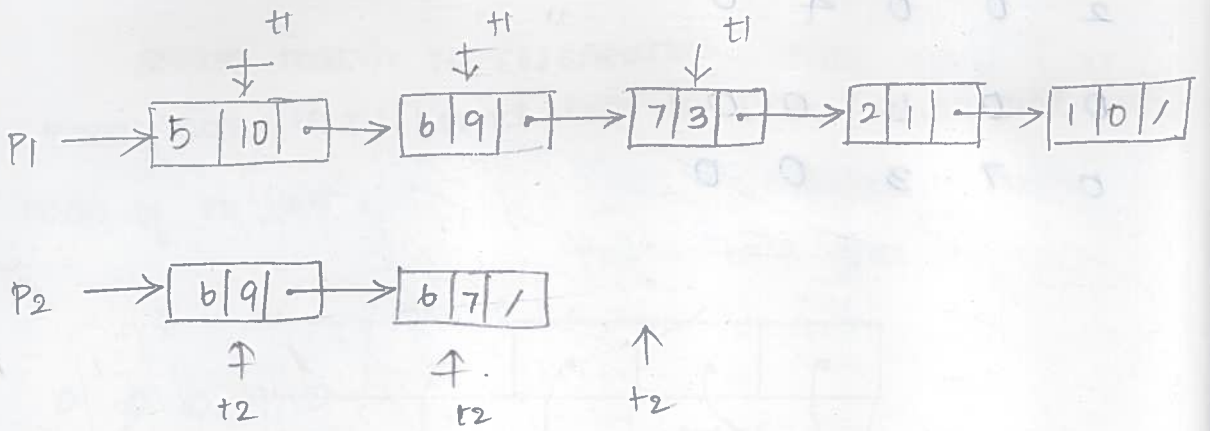
$5x^2 + 9x + 7$



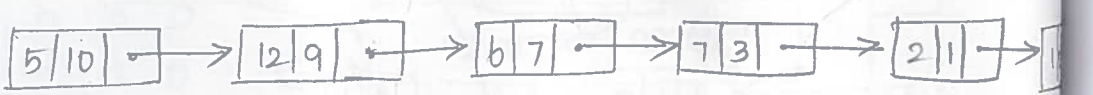
$$\begin{array}{r} 5x^2 + 4x + 2 \\ 5x + 5 \\ \hline 5x^2 + 9x + 7 \end{array}$$

$5x^{10} + 6x^9 + 7x^3 + 2x + 1$

$6x^9 + 6x^7$



Result :-



$\Rightarrow 5x^{10} + 12x^9 + 6x^7 + 7x^3 + 2x + 1$

Algorithm polyadd (List1, List2, List) :-

Begin

P1 = List1

P2 = List2

P = NULL

Ptail = NULL

while P1 != NULL AND P2 != NULL:

{ if (P1 → pow > P2 → pow)

{ addterm (P, Ptail, P1 → pow, P1 → coeff)

P1 = P1 → next

}

else if (P1 → pow < P2 → pow)

{

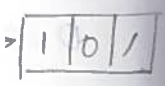
addterm (P, Ptail, P2 → pow, P2 → coeff)

P2 = P2 → next

}

End

```
else {
    if (P1->coeff + P2->coeff != 0)
        { addterm ( P, Ptail , P1->pow , P1->coeff + P2->coeff)
        }
    P1 = P1 ->next
    P2 = P2->next
  }
}
```




```
if
 P2 != NULL :
   P1 = P2

while  P1 != NULL :
 {
     addterm ( P , Ptail , P1->pow, P1->coeff)
     P1 = P1 ->next;
 }

List = P
  return List
End .
```

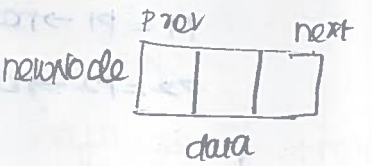

```
gorithm  addterm ( P, Ptail , power , coeff)
egin
     newNode = getnode (NODE)
     newNode -> pow =power   newNode -> coeff = coeff
    newNode -> next = NULL
     IF  (P == NULL)
          P=Ptail = newNode
       else {
            Ptail ->next = newNode
            Ptail =>newNode }
End
```

Exercises

Exercises on DLL :-

1. Insert a node into a sorted DLL.

Algorithm insert_sorted_DLL (Head, elt)



Begin

```
newNode = getNode (NODE)
newNode -> data = elt

if Head == NULL
{
    newNode -> PREV = NULL
    newNode -> next = NULL
     Head = newNode
      return Head
}

if (elt < Head -> data)
{
      newNode -> prev = NULL
      newNode -> next = Head
      Head -> prev = newNode
        Head = newNode
        return Head
}
temp = Head
while ( temp -> next -> data < elt)
          temp = temp -> next
temp -> next -> prev = newNode
newNode -> next = temp -> next
```

```
        temp -> next = newNode
        newNode -> prev = temp

        return Head

    End
```

2. Insert node after the given element in a DLL.

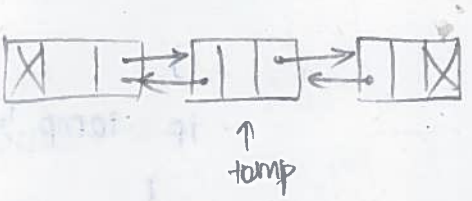
```
Algorithm  insert_after_elt ( Head, elt , val)
Begin
    if Head == NULL
        print "Empty List"
        return
    temp = Head
    while ( temp -> data != elt And temp != NULL)
        temp = temp -> next
    if temp != NULL
    {
        newNode = getnode (NODE)
        newNode -> data = value
        newNode -> next = temp -> next
        newNode -> prev = temp
        if temp -> next != NULL
            temp -> next -> prev = newNode
        temp -> next = newNode }
    else {
        print " Error, invalid Element " }

    return Head
End
```

3. Insert node before the given element in a DLL.

```
Algorithm insert_before_dll(Head, elt, value)
Begin
    if Head == NULL
        print Empty List
        return
    if Head -> data == elt
    { newNode = getnode(NODE)
      newNode -> data = value
      newNode -> prev = NULL
      newNode -> next = Head
        Head = newNode
    }
      temp = Head -> next
    while (temp -> data == elt and temp != NULL)
      { temp = temp -> next
      }
    if temp != NULL
     { newNode = getNode(NODE)
       newNode -> data = value
       newNode -> next = temp
         newNode -> prev = temp -> prev
       temp -> prev -> next = newNode
         temp -> prev = newNode
     } else
       { print "Error, invalid element" }
    return Head
End
```

4. Delete given element in DLL.

```
Algorithm deletekey DLL ( Head, key):
Begin
    if ( Head == NULL)
        return          print ("Empty list")

    if  Head -> data == key
    {   temp = Head
        Head = Head -> next
        Head -> prev = NULL
        free (temp) }

    temp = Head
    while ( temp != NULL and temp -> data != key)
            temp = temp -> next

    if temp -> next != NULL
    {
        temp -> prev -> next = temp -> next
        temp -> next -> prev = temp -> prev
        free(temp)
    }
    else
    {
        temp -> prev ->
        next = NULL
        free (temp)
    }
    return Head
End
```



5. Delete a given element in a sorted DLL.

Algorithm Delete key sorted DLL (Head, key)

Begin

```
if Head == NULL
    print "Empty List"  return

if key < Head->data
    print "key element not present in the list"
    return.
temp = Head
while (key  temp->data and temp
if key == Head->data
{   temp = Head
    Head -> next = Head
    Head -> prev = NULL
    free (temp) }
temp = Head -> next
while ( temp != NULL and key > temp->data)
{
    temp = temp -> next
}
if temp != NULL
{
    if temp -> next != NULL
    {
        temp -> prev -> next = temp -> next
        temp -> next -> prev = temp -> prev
        free (temp)
    }
```

```
        else
        {
            temp -> prev -> next = NULL
            free(temp)
        }
    }
    else
    { print "Key not found in the list"
      return }

    return Head

End.
```

Delete the node in a given position in DLL.

```
Algorithm deletepos DLL Head, pos)
Begin
    if Head == NULL
        print "Empty list"
        return

    if pos == 1
    { temp = Head
      Head -> next = Head
      Head -> prev = NULL
      free(temp) }

    temp = Head -> next    count = 2

    while (pos != count and temp != null)
    { temp = temp -> next
      count = count + 1
    }
```

```
if temp != null
{
    if temp->next != null
    {
        temp->prev->next = temp->next
        temp->next->prev = temp->prev
        free(temp)
    }
    else
    {
        temp->prev->next = NULL
        free(temp)
    }
}
else
{ print "position not available"
    return }

return Head

End
```

Exercise on circular linked List.

1. Delete first node of the circular singly linked list.

```
Algorithm delete first CLL (Head)
Begin
    if Head == NULL { print Empty list
                      return }
    temp = Head
    Head = temp->next
    temp2 = Head
    while (temp2->next != temp)
        temp2 = temp2->next
```

temp2 → next = Head

free (temp)

End.

. Delete last node of circular linked singly list.

Algorithm deleteLastCll (Head)

Begin

if Head == NULL

{ print Empty list

return }

if Head → next == Head

{ free (Head)

return }

prev = NULL

temp = Head

while (temp → next != Head)

{ prev = temp

temp = temp → next }

prev → next = Head

free (temp)

return Head

End.

02.01.2023 NON LINEAR DATA STRUCTURES. (CO4)

TREES

Hierarchical collection

partitioning of set into disjoint sets.

Ex :-

File directory structure (*) (level of Hierarchy.)

organization chart

moves in a game

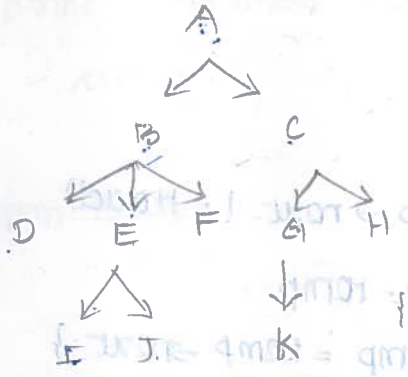
classification of hierarchies

Tree :-

tree is a set of nodes. that is

a. an empty set of nodes, or

b. has one node called the root from which zero or more trees (subtrees) descend.



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

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

{A}, {B} {D} {E, I, J} {F}

terminologies :-

tree is a collection of elements (nodes)

Each node may have 0 or more successors.

successors of A ⇒ B, C, D.

Each node has exactly one predecessor.

Links from node to its successor are called branches

successor of a node → children

predecessor of a node → parent

Nodes with same parent → siblings

Nodes with no children are called → leaves

Note :

A tree with N nodes has always N-1 edges

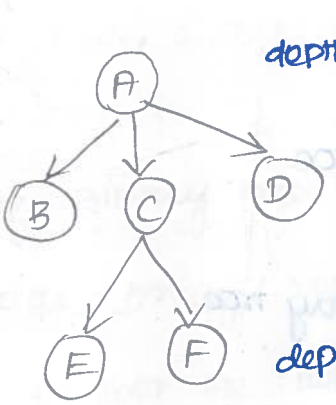
Two nodes in a tree incomplete.

Length of a path = number of edges

Depth of a node N = length of path from root to N.

Height of a node N = length of the longest path from N to a leaf.

Height of tree = height of a root.



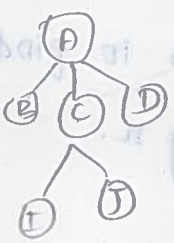
depth = 0, height = 2

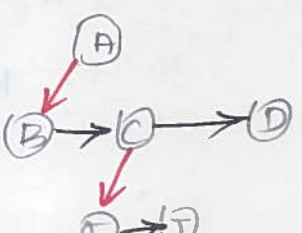
depth = 2, height = 0

Implementation :

pointer-based implementation } : Node with value and pointers to children.

child-sibling representation } Each node has 2 pointers: one to its first child and one to next sibling.





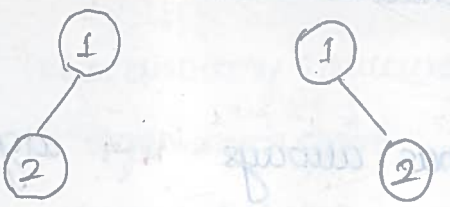
↓ - child pointer

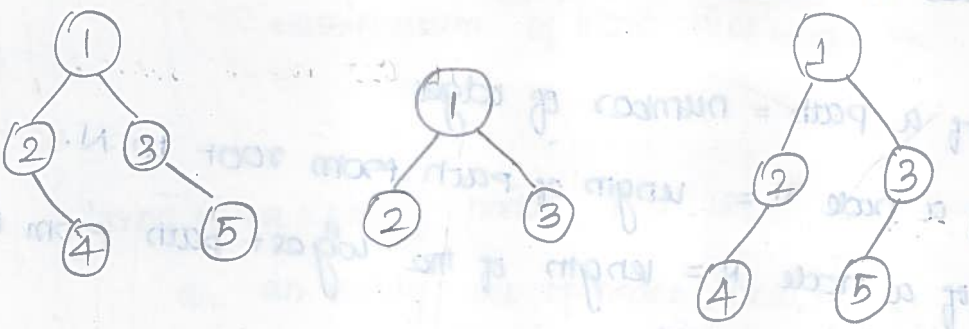
→ - sibling pointer

Binary trees :-

trees with number of children limited to maximum of 2.

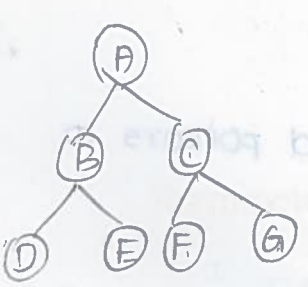
Root, left subtree & right subtree



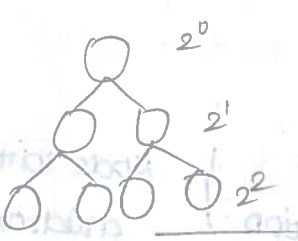


Types of Binary trees :-

- complete Binary tree
- skew tree
- strictly (or FULL) Binary tree.



Except leaf nodes all other nodes have exactly two children.

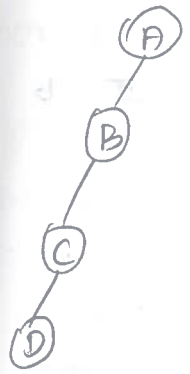


$2^0 + 2^1 + 2^2$

$\hookrightarrow 2^{h+1} - 1.$

when height is given to find no of nodes $= 2^{h+1} - 1$

when nodes is given to find height $= \log n$.



node given, height = node

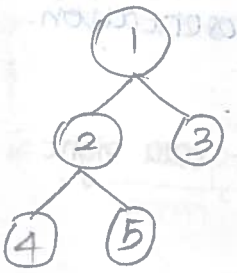
height given, node = height.

longest Binary tree for no of nodes.

Except leaf node all other nodes have exactly one children. ⇒ skew tree

if all nodes has left subtree ↑ → left skew

if all nodes has right subtree → right skew



⇒ strictly binary tree

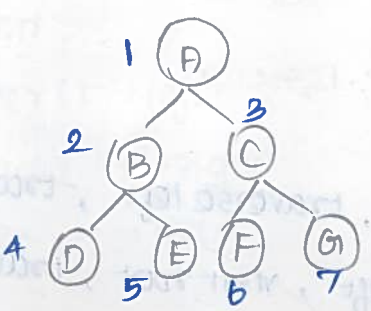
Either 0 (or) 2 children.

Representation of Binary trees:

⇒ sequential / Array representation

Number the nodes from level by level from left to right.

sume tree should be mplete binary tree



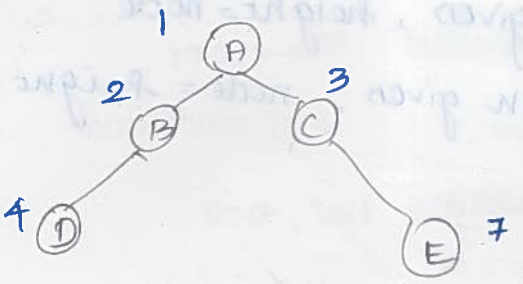
1	2	3	4	5	6	7
A	B	C	D	E	F	G

Given index.

parent of $i = \lfloor i/2 \rfloor$

left child of $p = 2p$

right child of $p = 2p+1$.

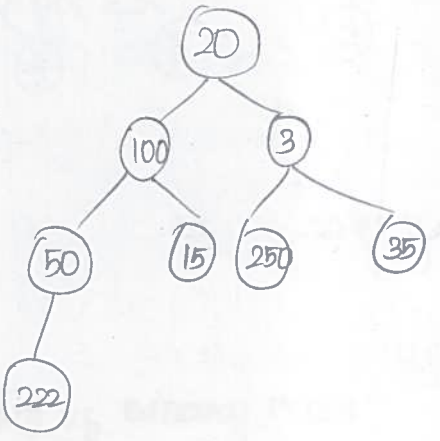


1	2	3	4	5	6	7
A	B	C	D	–	–	E

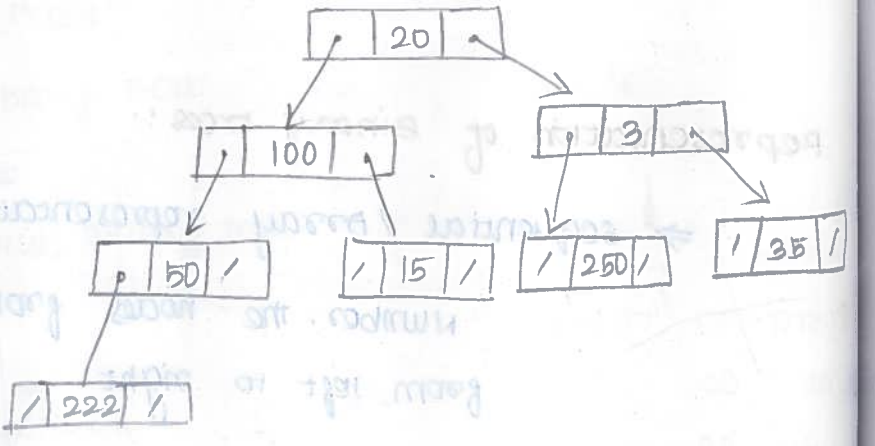
This representation is better for complete binary tree.

↳ worst for skew tree

⇒ Linked representation.



Representation.



Binary tree traversals :-

preorder : visit root, traverse left, traverse right

Inorder : traverse left, visit root, traverse right

post order : traverse left, traverse right, visit root

Inorder Traversal :-

```
Algorithm  inorder(t) :-
Begin
        if (t== NULL)
          return .
        inorder (t->left)
         print t->data.
        inorder (t->right)
         return .
End
```

Preorder Traversal :-

```
Begin
      if (t==NULL)
         return
      print t->data
       preorder (t->left)
        preorder (t->right)
         return
End
```

Postorder Traversal :-

```
Begin
       if(t==NULL)
         return
       postorder (t->left)
        postorder (t->right)
         print t->data
          return
  End.
```

3/8/2022

Binary Tree traversals :-

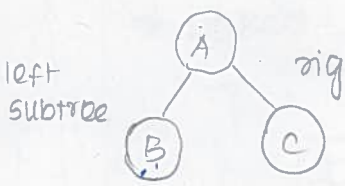
reference to the root node

Inorder traversal

Pre order traversal

Post order traversal.

order of processing based on left & right part of the root node.



Inorder :- Left ROOT Right

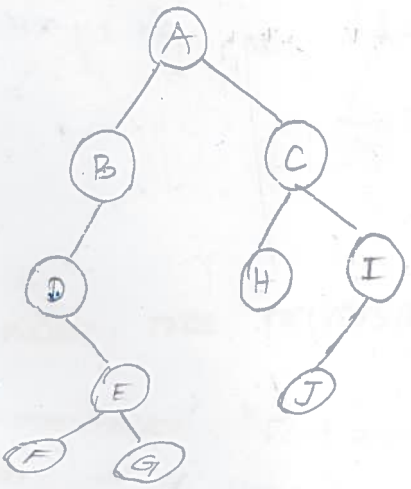
(BAC)

Preorder :- ROOT Left Right

(ABC)

Post order :- Left Right ROOT

(BCA)

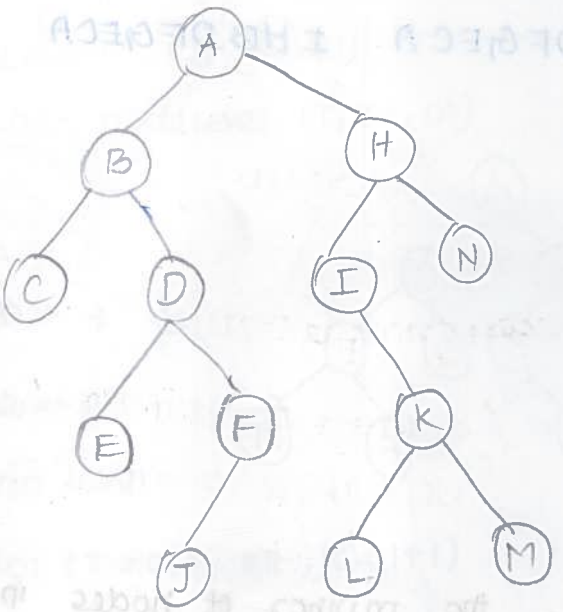


left root right

Inorder :- DFEGBAHCJI

Preorder :- ABDEFGCHIJ

Postorder :- FGEDBHJICA



Inorder :: C B E D J F A I L K M H N

Preorder :: A B C D E F J H I K L M N

postorder :: C E J F D B L M K I N H A

order to Tree conversion ::

only with inorder we cannot able to construct the proper Binary tree.

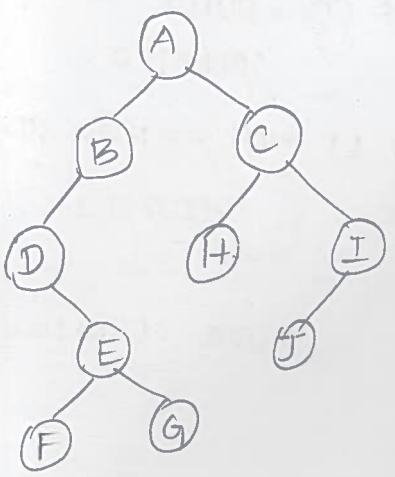
only if they given with inorder & pre order (or) inorder & post order. ⇒ then only we can able to construct the unique tree.

Inorder ::

D F E G B A H C I J

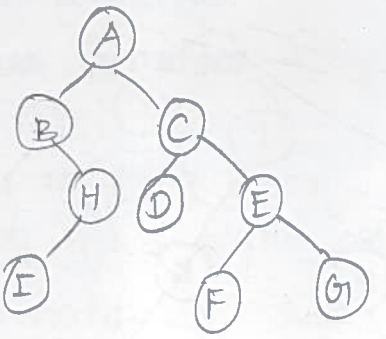
Preorder ::

A B D E F G C H I J



Inorder : I H B D C F E G

Postorder : I H D F G E C A



4/1/23

Algorithm to compute the number of nodes in a Binary tree:

```
Algorithm size(T) :-
Begin
    if (T = NULL)
        return 0
    else
        return size(T->left) + 1 + size(T->right)
End
```

no of node in left subtree — root node — no of nodes in right subtree

count no of leaf nodes.

```
Algorithm countleaf (T) :-
Begin
    if (T == NULL)
        return 0
    if (T->left == NULL and T->right == NULL)
        return 1
    else
        return countLeaf(T->left) + countleaf(T->right)
End
```

→ Here we should not add 1 because root node can be a leaf node in a tree.

Find a level of node with a given value :

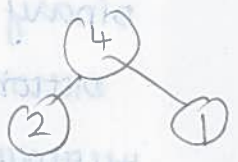
Algorithm nodeLevel(T, elt, level)

[Initial call nodeLevel (T, elt, 0)]

```
Begin
    IF T== NULL
        return -1

    if (T->data == elt)
        return level

    l = nodeLevel (T->left, elt, level+1)

    if (l != 0)
        return l

    else
        return nodeLevel (T->right, elt, level+1)

End
```

the no of even numbers stored in a binary tree.

Algorithm countevennodes (T, count)




```
Begin
    if T== NULL
        return 0

    if T->data % 2 == 0
        return count + countevennodes(T->left) + countevennodes (T->right)

    count += countevennodes(T->left) + countevennodes(T->right)
    return count
End
```

down.

Structurally identical trees :-

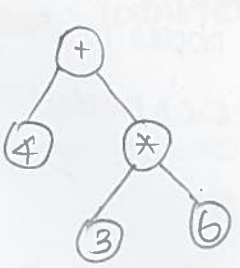
```
Algorithm identicalTree (T1, T2)
Begin
    If (T1 == NULL and T2 == NULL)        Both trees are Empty.
        return TRUE
    else if (T1 != NULL and T2 != NULL)   Both trees are non Empty
        if (T1->data == T2->data and
            identicalTree (T1->left, T2->left) and
                identicalTree (T1->right, T2->right))
            return TRUE
    else
        return FAISE
End
```

Applications :-

Expression trees
Binary search trees
Dictionary
Huffman trees.

Expression Trees :-

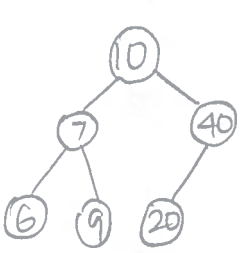
4 + 3 * 6



Internal nodes - operators
Leaf nodes - operands

Binary search trees :-

10, 7, 6, 40, 20, 9



Basic definition

terminologies

representation

traversal → tree → traversal

traversal → tree.

tree → array (or) linked list

array (or) linked list → tree.

Algo for traversal.

simple Algo on BT.

Array Programs:

Peak Element.

An element is called a peak element if its value is not smaller than the value of its adjacent Elements (if they exists).

Given an array arr[] of size N, find the index of any one of its peak elements.

Program:

```
n= int(input(" Enter the no of Elements : "))
Arr = []
for i in range(n):
    print('Number [',i,'I : ', end ='')
    temp = int(input())
    Arr.append(temp)
print('Array :', Arr)
print('Peak Elements :")
if n==1 or Arr[0] >= Arr[1]:
   print' Index value : 0 Element : ', Arr[0])
if  Arr[n-1] >= Arr[n-2]:
   print('Index value :',n-1,' Element :', Arr[n-1])

for i in range(1, n-1):
    if Arr[i] >= Arr[i-1] and Arr[i] >= Arr[i+1]:
        print(' Index value :', i, 'Element :', Arr[i])
```

Output:

```
Enter the no of Elements : 5
Number [0] : 1
Number [1] : 4
Number [2] : 2
Number [3] : 5
Number [4] : 3
```

```
Array : [1, 4, 2, 5, 3]
Peak Elements :
 Index value : 1 Element : 4
 Index value : 3 Element : 5
```

2. K^{th} Smallest & K^{th} Largest Element in an array.

Given an array arr = [] and an integer K where K is smaller than size of array. the task is to find the K^{th} smallest element in the given array. It is given that all array elements are distinct.

Program :

```
n = int(input('Enter the number of Elements : '))
Arr = []
for i in range(n):
    print('Number [', i, '] :', end = '')
    temp = int(input())
    Arr.append(temp)
print('Array :', Arr)
K = int(input('Enter the value of K :'))
Arr.sort()
print('Sorted Array :', Arr)
print('Kth smallest Element :', Arr[K-1])
print('Kth Largest Element :', Arr[n-K])
```

Output :

```
Enter the number of Elements : 5
Number [0] : 3
Number [1] : -1
Number [2] : 0
Number [3] : 2
Number [4] : -4
Array : [3, -1, 0, 2, -4]
Enter the value of K : 2
Sorted Array : [-4, -1, 0, 2, 3]
Kth smallest Element : -1
Kth Largest Element : 2
```

Sort an array with elements 0s and 1s

Sample input [1,0,0,1,1,0,1,0]

Sample output [0,0,0,0,1,1,1,1]

gram :

```
n = int(input('Enter the number of Elements: '))
Arr = []
count_0 = 0
for i in range(n):
    print('Number [',i,']: ', end='')
    temp = int(input(''))
    if temp == 0:
        count_0 += 1
    Arr.append(temp)
print('Array : ', Arr)

Arr2 = []
for i in range(n):
    if i < count_0:
        Arr2.append(0)
    else
        Arr2.append(1)

print('Array after sorting : ', Arr2)
```

output :

```
Enter the number of Elements : 8
Number [0]: 1
Number [1]: 0
Number [2]: 0
Number [3]: 1
Number [4]: 1
Number [5]: 0
Number [6]: 1
Number [7]: 0
Array : [1,0,0,1,1,0,1,0]
Array after sorting : [0,0,0,0,1,1,1,1]
```

4. Sort an array of 0s, 1s and 2s

Given an array of size N containing only 0s, 1s and 2s; Sort the array in ascending order.

Program:

```
n = int(input('Enter the no of Elements :'))
count_0 = 0
count_1 = 0
Arr = []
for i in range(n):
    print('Number [', i, ']: ', end = ' ')
    temp = int(input())
    if temp == 0:
        count_0 += 1
    elif temp == 1:
        count_1 += 1
    Arr.append(temp)
print('\n Array :', Arr)
Arr2 = []
for i in range(n):
    if i < count_0:
        Arr2.append(0)
    elif i < (count_0 + count_1):
        Arr2.append(1)
    else:
        Arr2.append(2)
print('\n Array after sorting :', Arr2)
```

Output:

```
Enter the no of Elements :
Number [0]: 0
Number [1]: 1
Number [2]: 2
Number [3]: 2
Number [4]: 0
Number [5]: 1
Array : [0, 1, 2, 2, 0, 1]
Array after sorting : [0, 0, 1, 1, 2, 2]
```

Move all negative elements to an end.

Given an unsorted array arr[] of size N having both negative and positive integers. The task is place all negative element at the end of array without changing the order of positive element and negative element.

Program:

```
n = int(input('Enter the no of Elements :'))
Arr = []
for i in range(n):
    print('Number [', i, '] :', end='')
    temp = int(input())
    Arr.append(temp)
print('Array :', Arr)
Arr2 = [0]*n;
p = 0
q = n-1
for k in range(n):
    if Arr[k]>=0:
        Arr2[p] = Arr[k] ; p+=1
    else:
        Arr2[q] = Arr[k]
        q-=1
Arr2[p:] = Arr2[n-1:p-1:-1]
print('\n Re-Arranged Array :', Arr2)
```

Output:

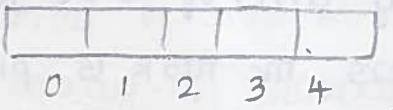
```
Enter the no of Elements : 6
Number [0] : 9
Number [1] : -8
Number [2] : 1
Number [3] : 0
Number [4] : -3
Number [5] : -1
Array : [9, -8, 1, 0, -3, -1]
```

Re-Arranged Array :

[9, 1, 0, -8, -3, -1]

ADDRESSING FUNCTIONS :

1D Array :

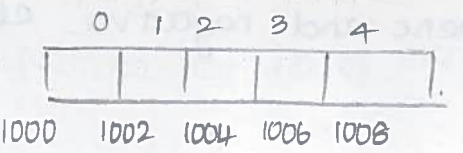


Data type : Integer

↓

size = 2

Base address : 1000



Addressing Function :

Address (data[i]) = Base address +

size * (index)

2D Array : ROW = 3 COL = 3

00	01	02
1	2	3
10	11	12
4	5	6
20	21	22
7	8	9

2D array can be represented as 1D array in two different ways :

(i) Row major order

(ii) Column major order .

Row major order :

0	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
00	01	02	10	11	12	20	21	22

Addressing Function :

Address = Base address +

size (no of elements in preceding row + no preced

elements in

Address (data[i][j]) =

Base address + size (i * cols + j)

column major order :

0	1	2	3	4	5	6	7	8
1	4	7	2	5	8	3	6	9
00	10	20	01	11	21	02	12	22

Addressing Function :

Address (data[i][j]) =

Base address + size * (j * rows + i)

3D Array :

ROW = i COL = j slice = k.

row major order :

Address (data[i][j]) = Base address +

size (~~i~~ i * cols + j)

↓

k * rows * cols +

column major order :

Address (data[i][j]) = Base address +

size (k * rows * cols + j * rows + i)

i j k

↑ ↑ ↑

A[3][3][3]

BA = 100

A[0][1][2] =

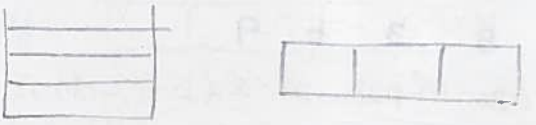
= 100 + 2((2*3*3) + (0*3+1))

= 100 + 2 (18 + 1)

= 100 + 2(19) = 100 + 38

= 138

STACK.



Initially top value pointed top = -1.

```
Algorithm   push ( stack , n, top, element) :
  Begin
      if  top = = n-1 :
          print ( " stack is Full ")
      else     top = top+1
          stack [top] = element
  End.

Algorithm   Pop ( stack , top) :
Begin
      if top = = -1 :
          print ( " stack is Empty ")
      else
          print ( stack [top])
          top = top - 1.
End .
```

Write an Algorithm using stack to convert an Infix Expression to postfix Expression.

Trace the Algorithm on input a*(b+c-d)/e (12m)

2. Write an Algorithm using stack to check the well formness of paranthesis.

Trace the Algorithm on input {[(a+b)-d]*e} (8m).

3. Write Algorithm for Insert and delete an element from circular queue

1. Infix to Postfix Conversion :- a*(b+c-d)/e

Infix Expression	stack content	Postfix Expression
a*(b+c-d)/e		a
*(b+c-d)/e	*	a
(b+c-d)/e	*(	a
b+c-d)/e	*(	ab
+c-d)/e	*(+	ab
c-d)/e	*(+	abc
-d)/e	*(- (= ⊖) (⊕ → POP)	abc+
d)/e	*(-	abc+d
)/e	*(- (① pop everything until open '(' ; discard)	abc+d-
/e	* (① → POP)	abc+d-*
e	/	abc+d-*e
Infix - Empty.	① pop all	ab*c+d-*e/