Subject Name : Data Structure

DS DAY-01:

Q. Why there is a need of data structure in programming?

There is a need of data structures in programming to achive efficiency in operations.

Q. What is a data structure?

It is a way to store data elements into the memory (i.e. into the main memory) in an **organized manner** so that operations like **addition, deletion, searching, sorting, traversal etc....** can be performed on it efficiently.

- we want to store marks 100 students:

int m1, m2, m3, m4, ..., m100;//sizeof(int): 4 bytes => 400 bytes

- we want to arrange marks in a descending order => sorting

int arr[100];//400 bytes

+ Array:

An array is **linear/basic data structure**, which is a **collection/ list of logically related similar type of data elements** gets stored into the memory at **contiguos locations.**

int arr[5] ;

- in an array, to covert array notation into its eq pointer notation is done by the compiler,

i.e. to maintained link between array elements in an array is the job of compiler.

- we want to store records of 100 employees:

empid : int
name : char [ ] / string
salary : float

- in an array we can collect/combine logically related similar type of data elements only, and hence to overcome this limitation structure data structure has been designed.

+ Structure:
It is a **linear / basic data structure**, which is a **collection / list of logically related similar and disimmilar type of data elements** gets stored into the memory **collectively as a single entity / record.**

```
struct employee
{
        int empid;//4 bytes
        char name[ 32 ];//32 bytes
        float salary;//4 bytes
};
```

sizeof structure = sum of size of all its members
sizeof(struct employee): 40 bytes

- there are 2 types of data types:
1. primitive/predefined/builtin data type: char, int, float, double, void
=> data types which are already known to the compiler

2. non-primitive / derived / user defined data type : pointer, array, structure, union, enum, function etc...
=> data types which are not already known to the compiler, i.e. need to define/derived by the programmer

user defined data type:
typedef

```
typedef int INT;//user defined => not a derived data type
typedef int bool_t;//user defined => => not a derived data type
typedef struct employee emp_t;//derived / user defined
int *iptr;//derived data type
```

- there are two types of data structures:

1. linear / basic data structures: data structures in which data elements gets stored into the memory in a linear manner/linearly and hence can be accessed lineary (i.e. one after another).
- array
- structure & union
- linked list
- stack
- queue

2. non-linear / advanced data structures: data structures in which data elements gets stored into the memory in a non-linear manner (e.g. hierarchical manner), and hence can be accessed non-linearly.
- tree
- graph
- binary heap
- hash table

etc....

structure is a derived data type =>

```
struct employee
{
        int empid;//4 bytes
        char name[ 32 ];//32 bytes
        float salary;//4 bytes
};
```
=> compiler

<data type> <var_name>;
int num;

struct employee emp;

- to learn data structures, is not to learn any programming language, it is nothing but to learn algorithms, data structure algorithms can be implemented in any programming langauge (C / C++ / Java / Python).

Q. What is a Program? => Machine
- A Program is a finite set of instructions written in any programming language (i.e. either in low level / high level) given to the machine to do specific task.

Q. What is an algorithm? => User/Human Beings
- An algorithm is a finite set of instructions written in any human understandable langauge like english, if followed, acomplishesh given task.

- Program is an implementation of an algorithm.
- An algorithm is like a blue print / design of a program on paper.

Blue-print => implementation => building

- Pseudocode is a special form of an algorithm for programmer user.
Q. What is a pseudocode? => Programmer User
- An algorithm is a finite set of instructions written in any human understandable langauge like english with some programming constraints, if followed, acomplishesh given task, this kind/form of an algorithm us referred as a pseudocode.
- an algorithm to do sum of array elements:
- traversal on array => to visit each array element sequentially from first element max till last element.

algorithm:
step-1: initially take sum as 0.
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final sum.

```
Pseudode:  => Programmer
Algorithm ArraySum( arr, n){//arr is an array of size n
        sum = 0;
        for( index = 1 ; index <= n ; index++ ){
                sum += arr[ index ];
        }
        return sum;
}


Program: => Machine
int array_sum( int arr[ ], int size ){

        int sum = 0;
        int index;
        for( index = 0 ; index < size ; index++ ){
                sum += arr[ index ];
        }
        return sum;
}
```

Example:
IT Industry:
Client (Algorithm i.e. Requirement) => Software Architect/Tech Manager
Software Architect (Pseudocode) => Software Developer
Software Developer (Program) => Machine.

- an algorithm is a solution of a given problem.
- an algorithm = solution
- one problem may has many solutions
e.g.
searching => to search/find an element (let say referred as key element), in a given collection/list/set of elements.
1. linear search
2. binary search

sorting => to arrange data elements in a collection/list of elements wither in an ascending order (or in a descending order).
1. selection sort
2. bubble sort
3. insertion sort
4. merge sort
5. quick sort

etc.....

- if one problem has many solutions, we need to select an efficient solution out of them, and to decide which solution/algorithm is an efficient one, we need to do their analysis.

- **analysis of an algorithm** is a work of calculating/determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.
- there are 2 measures of analysis of an algorithm:
1. time complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. space complexity of an algorithm is the **amount of space i.e. computer memory** it needs to run to completion.

1. linear search / sequential search:
algorithm:
step-1: accept key from user (key = element which is to be search)
step-2: start traversal of an array from first element and compare value of key element with each array element sequentialy till match is not found or max till last element.
step-3: if the value of key is matches with any of the array element then return true other wise return false.

pseudocode:

```
Algorithm LinearSearch(A, key, n)
{
        for( index = 1 ; index <= n ; index++  ) {
                if( key == A[ index ] )//if key matches with any array ele
                        return true;//key is found
        }

        //if key do not matches with any of array element
        return false;//key is not found
}
```

best case occurs : **if key is found at first position => O(1)**
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 50 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1

.
.
.
if size of an array = n => no. of comparisons = 1

worst case occurs : if either key is found at last position or key does not exists : **O(n).**

if size of array = 10 => no. of comparisons = 10
if size of array = 20 => no. of comparisons = 20
if size of array = 100 => no. of comparisons = 100
.
.
if size of array = n => no. of comparisons = n

best case time complexity = if an algo takes min amount of time to run to completion.

worst case time complexity = if an algo takes max amount of time to run to completion.

average case time complexity = if an algo takes neither min nor max amount of time to run to completion.

+ Asymptotic analysis: it is a **mathematical** way to calculate time complexity and space complexity of an algorithm without implementing it in any programming language.
- in this kind of analysis, focus is on basic operation in that algorithm
e.g. searching => basic operation is comparison, and hence anlaysis can be done depends on no. of comparisons takes places in different cases.
sorting => basic operation is comparison, and hence anlaysis can be done depends on no. of comparisons takes places in different cases.
Addition of matrices => basic operation addition, and hence analysis can be done depends on number addition instructions.

- there are some notations and few assumptions that we need to follow:

there are 3 asymtoptic notations:
1. Big Omega (Ω) - this notation is used to represent **best case time complexity.**
- big omega is referred as asymtptic lower bound
- running of an algo should not be less than its **asymtotic lower bound.**

2. Big Oh (O) - this notation is used to represent **worst case time complexity**
- big oh is referred as asymtptic upper bound.
- running of an algo should not be greater than its asymtotic upper bound.

3. Big Theta (θ) - this notation is used to represent an **average case time complexity.**
- big theta is referred as asymtptic tight bound.

Assumption:
- if running time of an algo is having additive / substractive / multiplicative / divisive constant it can be neglected.
e.g.
O(n + 3) => O(n)
O(n – 5) => O(n)
O(n / 3) => O(n)
O(2*n) => O(n)

- if an algo follows divide-and-conquer approach then we get time complexity in terms of log.

DS DAY-02:

2. Binary Search
algorithm:
step-1: accept key from user
step-2:
- calculate mid pos by the formula => mid = (left+right)/2
- by means of calculating mid position, big size array gets divided logically into two subarray's => left subarray & right subarray
- left subarray is from left to mid-1, and right subarray is from mid+1 to right.

for left subarray => value of left remains as it is, value of right = mid-1
for right subarray => value of right remains as it is, value of left = mid+1

step-3: compare value of key with ele at mid pos, if key matches with ele at mid pos return true

step-4: if key do not matches then search key either into the left subarray or into the right subarray

step-5: repeat step-2, step-3 & step-4, till either key not found, or max till subaarray is valid, if subarray is invalid then return false indicates key not found.

if(left <= right) => subarray is valid
if(left > right) => subarray is invalid

- in a binary tree, any element is at either one of the following 3 positions:
1. root pos
2. leaf pos
3. non-leaf pos

root node/root pos => first pos
node which is not having further child node => leaf node
node which is having child node => non-leaf node

if key is found at root pos => best case => O(1)
no. of comparisons for input size array = 1
time complexity = Ω(1).

if key is found at non-leaf pos => average case
O(log n)
time complexity = θ(log n)

if either key is found at leaf pos or key is not found => worst case
O(log n)
time complexity = O(log n)

+ Sorting Algorithms:
Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.
- when we say sort array elements, bydeafult we need to sort array elements in an ascending order.

1. Selection Sort:

for size of an array is n,
in iteration-1: no. of comparisons = n-1
in iteration-2: no. of comparisons = n-2
in iteration-3: no. of comparisons = n-3
.
.

iterations (n-1):

total no. of comparisons = (n-1) + (n-2) + (n-3) + ...
total no. of comparisons = n(n-1)/2 => (n^2 – n) / 2

T(n) = O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

rule/assumption: if running time of an algo is having a polynomial, then in its time complexity only leading term will be considered.
e.g.
O(n^3 + n + 4) => O(n^3)
O(n^2 + 5) => O(n^2)
O(n^3 + n^2 + n – 3) => O(n^3)

DS DAY-03:
2. Bubble Sort:

for size of an array is n,
in iteration-1: no. of comparisons = n-1
in iteration-2: no. of comparisons = n-2
in iteration-3: no. of comparisons = n-3
.
.

iterations (n-1):

total no. of comparisons = (n-1) + (n-2) + (n-3) + ...
total no. of comparisons = n(n-1)/2 => (n^2 – n) / 2

T(n) = O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

```
for( pos = 0 ; pos < SIZE-it-1 ; pos++ )
```

for it=0; pos=0,1,2,3,4
for it=1; pos=0,1,2,3
for it=2; pos=0,1,2
.
.

best case:
input array => 10 20 30 40 50 60
flag= 0

for it=0
iteration-0:
10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

for it=1
iteration-1: all next iterations will be skipped

- if all pairs are in order
=> if there is no need of swaaping in first iteration, it means all pairs are already in order => array is already sorted -> no need to go to next iteration and we are reducing no. of comparisons at an implementation level.

=> array ele's are already sorted

- in best case this algo takes only one iteration and
total no. of comparisons = n-1
T(n) = O(n - 1) => O(n)
best case time complexity = Ω(n).

3. Insertion Sort:

```
for( i = 1 ;  i < SIZE ; i++ ){
        j = i-1;
        key = arr[ i ];

        while( j >= 0 && key < arr[ j ] ){
                arr[ j+1 ] = arr[ j ];//shift ele towards its right by 1
                j--;//goto the pre element
        }

        //insert key into the left hand i.e. into an array at its appropriate pos
        arr[ j+1 ] = key;
}
```

best case : if array ele's are already sorted:

iteration-1: no. of iterations=1
10 20 30 40 50 60

10 20 30 40 50 60

iteration-2: no. of iterations=1
10 20 30 40 50 60

10 20 30 40 50 60

iteration-3: no. of iterations=1
10 20 30 40 50 60

10 20 30 40 50 60

iteration-4: no. of iterations=1
10 20 30 40 50 60

10 20 30 40 50 60

iteration-5: no. of iterations=1
10 20 30 40 50 60

10 20 30 40 50 60

insertion sort algo takes max (n-1) no. of iterations, and in each iteration only one comparison takes place, and hence
total no.of comparisons = 1 * (n-1) = n-1
T(n) = O(n – 1) => O(n) => Ω(n).

4. Merge Sort:
- this algo follows divide-and-conquer approach
- there are 2 steps in this algo:

step-1: divide big size array logically into smallest size subarray's i.e. subarray having size 1.

- in each iteration by means of calculting mid pos, we are dividing big size array logically into two subarray's, left subarray will be from left to mid, and right subarray will be from mid+1 to right.

for left subarray, value of left remains as it is and right = mid
for right subarray, value of right remains as it is and left = mid+1

step-2: merge two already sorted subarray's into a single array in a sorted manner.
- if subarray contains only 1 ele => subarray is already sorted.

 # DS DAY-04:
5. Quick Sort:
- this algo follows divide-and-conquer approach.
- the basic logic of this algo is partitioning

partitioning:
step-1: to select pivot element (in an array we can select either **leftmost** or rightmost or middlemost element as a pivot element)

step-2: shift all ele's which are smaller than pivot towards its left and shift all ele's which are greater than pivot towards its right, by means of this shifting pivot element gets fixed at its appropriate pos and big size array gets divided logically into two partitions, **left partition & right partition.**

step-3: apply partitioning on left partition as well as right partition till the size of partition is greater than 1.

- limitations of an array data structure:
1. in an array we can collect/combine logically related similar type of data elements only ==> to overcome thi limitation structure data structure has been designed.

2. **array is static** i.e. size of an array remains fixed, it cannot be either grow or shrink during runtime.
int arr[100];

3. **addition & deletion operations on an array are not efficient as it takes O(n) time.**
As while addition we need to shift ele's towards right and while deletion we need shift ele's towards left and depends on the size of an array i.e. n and hence addition & deletion operations on array takes O(n) time.

Why linked list?
- to overcome limitation 2 & 3 of an array linked list data structure has been designed.
- linked list must be dynamic, and addition & deletion operations must be efficient i.e. expected time complexity is O(1).

Q. What is a Linked List?
It is a **basic / linear data structure**, which is **a collection / list of logically related similar type of data elements** in which, **an addr of first element always gets stored into a pointer variable referred as head**, and **each element contains actual data and an addr of its next element (as well as an addr of its previous element).**
i.e. elements in this collection / list / data structure are explicitly linked with each other, and hence it is called as **linked list.**
- in linked list, element is also called as a node.

- basically there 2 types of linked list:
1. singly linked list : it is a type of linked list in which each element/node contains actual data and an addr of its next node i.e. each node contains only one/single link.
- further there are 2 types of singly linked list
i. singly linear linked list
ii. singly circular linked list

2. doubly linked list : it is a type of linked list in which each element/node contains actual data, an addr of its next node as well as an addr of its prev node i.e. each node contains two links.
- further there are 2 types of doubly linked list
i. doubly linear linked list
ii. doubly circular linked list

- there are total 4 types of linked list:
i. singly linear linked list
ii. singly circular linked list
iii. doubly linear linked list
iv. doubly circular linked list

i. singly linear linked list: it is a type of linked list in which,
- head always contains an addr of first node, if list is not empty
- each node has two parts:
1. data part : contains actual data if any primitive / non-primitive type
2. pointer part (next) : contains an addr of its next node
- last node points to NULL. i.e. next part of last node contains NULL.

Node/element in a slll has two parts:

```
struct node
{
	int data;//4 bytes
	struct node *next;//4 bytes => self referential pointer
};
```

sizeof(struct node) = 8 bytes (32-bit compiler)

to want store an addr of int variable => int *
to want store an addr of char variable => char *

.
.

to want store an addr of struct employee variable => struct employee *
to want store an addr of struct student variable => struct student *
to want store an addr of type variable => type *

sizeof(char *) = 4 bytes
sizeof(int *) = 4 bytes
sizeof(float *) = 4 bytes
sizeof(double *) = 4 bytes
sizeof(struct employee *) = 4 bytes

sizeof(type *) = 4 bytes

Q. What is a NULL?
NULL is a predefined macro whose value is 0 which is typecasted into a void *

#define NULL ((void *)0)

- on linked list data structure we can perform basic two operations:
1. addition : to add node into the linked list
2. deletion : to delete node from the linked list

1. addition : to add node into the linked list
- we can add node into the linked list by three ways:
1. add node into the linked list at last position
2. add node into the linked list at first position
3. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from the linked list by three ways:
1. delete node from the linked list at first position
2. delete node from the linked list at last position
3. delete node from the linked list at specific position (inbetween position)

1. add node into the linked list at last position (slll):
- we can add as many as we want number of nodes into slll in O(n)

Best Case : Ω(1)
Worst Case : O(n)
Average Case : θ(n)

traversal of the linked list : to visit each node in a linked list sequentially from first node max till last node.
- in a slll, we can always start traversal from first node

- general syntax to define an array by concept:
<data type> <arr_name>[size];

data type may any primitive / non-primitive type
arr_name is an identifier
[ ] => subscript operator
size => constant => size of an array

DS DAY-05:

2. add node into the linked list at first position (slll):
- we can add as many as we want number of nodes into slll in O(1)

Best Case : Ω(1)
Worst Case : O(1)
Average Case : θ(1)

- In linked list programming -> rule
make before break i.e. always creates new links first (links associated with newly created node), and then only break old links.

3. add node into the linked list at specific position (inbetween pos) (slll):
- we can add as many as we want number of nodes into slll in O(n)

Best Case : Ω(1) => if pos == 1
Worst Case : O(n)
Average Case : θ(n)

2. deletion : to delete node from the linked list
- we can delete node from the linked list by three ways:

1. delete node from the linked list at first position
- we can delete node which is first pos from slll in O(1) time.
Best Case : Ω(1) => if list contains only one node
Worst Case : O(1)
Average Case : θ(1)

2. delete node from the linked list at last position:
- we can delete node which is first pos from slll in O(n) time.
Best Case : Ω(1) => if list contains only one node
Worst Case : O(n)
Average Case : θ(n)

3. delete node from the linked list at specific position (inbetween position)
- we can delete node which is first pos from slll in O(n) time.
Best Case : Ω(1) => if pos == 1
Worst Case : O(n) => if pos == count_nodes( )
Average Case : θ(n)

+ limitation of singly linear linked list:
- we can traverse slll only in a forward direction
- prev node of any node cannot accessed from it
- add_last( ) & delete_last( ) functions are not efficient as it takes O(n)time.
- any node cannot be revisited => to overcome this limitation **scll** has been designed.

- operations/algorithms (operations/algorithms: add_last(), add_first( ), add_at_pos( ), delete_last( ), delete_first() and delete_at_pos( ))
that we applied on slll, all operations can be applied as it is onto scll as well, except we need to maintain next part of last node.

+ limitation of singly circular linked list:
- we can traverse scll only in a forward direction
- prev node of any node cannot accessed from it
- as even while adding or deleting node at first pos, we need to traverse list till last node to maintain next part of last node, and hence **add_last( ), add_first( ), delete_last( ) and delete_firrst( )** operations are not efficient as it takes O(n) time.

To overcome limitations of singly linked list (i.e. slll as well as scll), doubly linked list has been designed.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < \ldots\ldots\ldots$

Home Work:
menu driven program to implement scll functions.

- we have divided logic of a program into functions/procedure

- C programming Lanaguage is a procedure oriented programming langauge: in C programming logic of a program gets divided into the functions i.e. into procedures.

DS DAY-06:
doubly linear linked list: it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
1. data part: actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
3. pointer part (prev): contains an addr of its prev node
- next part of last node and prev part of first always points to NULL.

```
struct node
{
        int data;//4 bytes
        struct node *next;//4 bytes => self referential pointer
        struct node *prev;//4 bytes => self referential pointer
};
```

sizeof(struct node) = 12 bytes (32-bit compiler)

- operations/algorithms (operations/algorithms: add_last(), add_first( ), add_at_pos( ), delete_last( ), delete_first() and delete_at_pos( ))
that we applied on slll, all operations can be applied as it is onto dlll as well, except we need to maintain forward link and backward link of each node i.e. we need to maintained next part as well prev part of each node.

+ advantages of dlll:
1. dlll can be traversed in forward dir as well in a backward dir
2. in a dlll, prev node of any node can be accessed from it

add_last( ) => O(n)
add_first( ) => O(1)
delete_first( ) => O(1)
delete_last( ) => O(n)

+ disadvantages of dlll:
- add_last() & delete_last( ) operations are not efficient as it takes O(n) time
- traversal can be start only from first node.
And hence to overcome above limitations of dlll, dcll has been designed.
To overcome limitations of all first 3 types of linked list, dcll has beeb designed.
- dcll is most efficient form of a linked list.

doubly circular linked list: it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
1. data part: actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
3. pointer part (prev): contains an addr of its prev node
- next part of last node always contains an addr of first node and prev part of first always contains an addr of last node.

- operations/algorithms (operations/algorithms: add_last(), add_first( ), add_at_pos( ), delete_last( ), delete_first() and delete_at_pos( ))
that we applied on dlll, all operations can be applied as it is onto dcll as well, except we need to maintain forward link and backward link of each node i.e. we need to maintained next part as well prev part of each node and prev part of first node and next part of last node.

- in a dcll, as we can get addr of last node from head->prev in O(1) time, add_last( ), add_first( ), delete_last( ) & delete_first( ) all operations are efficient as it takes **O(1)** time.

+ advantages of dcll:
- dcll can be traversed in forward as well in a backward dir, and traversal can be started either from first node or from last node in O(1) time.
- addr of first node gets always from head in O(1) time
- addr of last node gets always from head->prev in O(1) time

- add_at_pos( ) & delete_at_pos( ) functions takes O(n) time, as we need to traverse list till (pos-1)th node.

Linked List <=> Doubly Circular Linked List
DCLL: dynamic
addition & deletion operations are effiecient in a linked list than in an array as it takes O(1) time.

+ Stack:
- it is a **basic / linear data structure**, which is **a collection / list of logically related similar type of data elements** in which elements can be added as well deleted from only one end referred as **top** end.
- in this collection / list, element which was inserted last can only be deleted first, so this list works in **last in first out** / **first in last out** manner and hence stack is also called as **lifo list / filo list.**

- we can perform 3 basic operations on stack data structure in O(1) time:
1. Push : to insert/add an element onto the stack from top end
2. Pop : to delete / remove an element from the stack which is at top end
3. Peek : to get the value of an element from the stack which is at top end.

- Stack can be implemented by 2 ways:
1. static implementation of stack by using an array
2. dynamic implementation of stack by using linked list

1. static implementation of stack by using an array:

```
struct stack
{
        int arr[ 5 ];
        int top;
};
```

sizeof(struct stack) = 24 bytes

arr   :   int [ ] => non-primitive
top   :   int   => primitive

1. Push : to insert/add an element onto the stack from top end
step-1: check stack is not full (if stack is not full then only we can push element onto it).
step-2: increment the value of top by 1
step-3: insert/push an element onto the stack at top end

2. Pop : to delete / remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop element from it).
step-2: decrement the value of top by 1
[by means of decrementing value of top by 1, we are achieving deletion of an element from the stack].

3. Peek : to get the value of an element from the stack which is at top end.
step-1: check stack is not empty (if stack is not empty then only we can peek an element from it).
step-2: get/return the value of an element from the stack which is at top end.
[without push/pop an element from the stack].

2. dynamic implementation of stack by using linked list (dcll):
push : add_last( )
pop : delete_last( )

OR

push : add_first( )
pop : delete_first( )

head -> 44 33 22 11

applications of stack data structure:
1. to control flow of an execution of a program an OS maintains stack.
2. in recursion internally an OS maintains stack
3. undo & redo functionalities of an OS has been implemented by using stack
4. stack is used to implement advanced data structure algorithm like **dfs depth first search traversal** in a tree & graph.
5. stack is used to implement expression conversion & evaluation algorithms:
- to convert given infix expression into its equivalent postfix expression
- to convert given infix expression into its equivalent prefix expression
- to convert given prefix expression into its equivalent postfix expression
- to evaluate postfix expression
etc...
- programs in which, list/collection of elements should works in a lifo manner data structure stack can be used.

DS DAY-07:
+ Stack Application Algorithms:
- to convert given infix expression into its equivalent postfix expression
- to convert given infix expression into its equivalent prefix expression
- to convert given prefix expression into its equivalent postfix expression
- to evaluate postfix expression

Q. What is an expression?
- An expression is a combination of an operands and operators.

- there are 3 types of expression:
1. infix expression : a+b
2. prefix expression : +ab
3. postfix expression : ab+

infix expression => 100 + 25 – 5 / 25
postfix:
=> 100 + 25 – 5 / 25
=> 100 + 25 – 5 25 /
=> 100 25 + - 5 25 /
=> 100 25 + 5 25 / -

=> 100 25 + 5 25 / -

cur ele = -
op1 = 125
op2 = 0
result = 125 - 0

stack:

125

Stack Application:

DS DAY-08:

+ Queue: it is a **basic / linear data structure**, which is **a collection / list of logically related similar type of data elements**, in which elements can be inserted from one end referred as **rear end**, and elements can deleted from another end referred as **front end.**
- In data structure queue, element which was inserted first can be deleted first, so this list works in **first in first out / last in last out** manner, and hence queue is also called as **FIFO List / LILO List.**
- We can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
2. dequeue : to delete/remove an element from the queue which is at from end.

- there are total 4 types of queue:
1. linear queue
2. circular queue
3. priority queue
4. double ended queue (deque)

1. linear queue: (fifo)

- we can implement linear queue by 2 ways:
i. static implementation of a linear queue (by using an array)
ii. dynamic implementation of a linear queue (by using linked list)

i. static implementation of a linear queue (by using an array):

```
struct queue
{
        int arr[ SIZE ];
        int rear;
        int front;
};

arr    : int [ ] => non-primitive
rear   : int   => primitive
front : int   => primitive
```

1. enqueue : to insert an element into the queue from rear end
step-1: check queue is not full (if queue is not full then only we can insert an element into it).
step-2: increment the value of rear by 1
step-3: insert an element into the queue from rear end
step-4: if(front == -1) then front = 0

2. dequeue : to delete/remove an element from the queue which is at from end.
step-1: check queue is not empty (if queue is not full then only we can delete an element from it).
step-2: increment the value of front by 1
[by means of incrementing value of front by 1 we are achieving deletion of an element from the queue].

+ limitation of a linear queue:
- vacant places cannot be reutlized
- to overcome this limitation circular queue has been designed

- effective memory utilzation can be acheived in a circular queue

circular queue:

rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2
rear = 2, front = 3
rear = 3, front = 4

if front is at next pos of rear => cir queue is full

front == (rear+1)
0 == (4+1)
0 == 5 => LHS != RHS

front == (rear+1)%SIZE

for: rear = 0, front = 1 => front is at next pos of rear => cir queue is full
=> front == (rear+1)%SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir queue is full.

for: rear = 1, front = 2 => front is at next pos of rear => cir queue is full
=> front == (rear+1)%SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir queue is full.

for: rear = 2, front = 3 => front is at next pos of rear => cir queue is full
=> front == (rear+1)%SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir queue is full.

for: rear = 3, front = 4 => front is at next pos of rear => cir queue is full
=> front == (rear+1)%SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir queue is full.

for: rear = 4, front = 0 => front is at next pos of rear => cir queue is full
=> front == (rear+1)%SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir queue is full.

rear++;
=> rear = rear + 1;

rear = (rear+1)%SIZE

for : rear=0 => rear = (rear+1)%SIZE => (0+1)%5 => 1%5 => 1
for : rear=1 => rear = (rear+1)%SIZE => (1+1)%5 => 2%5 => 2
for : rear=2 => rear = (rear+1)%SIZE => (2+1)%5 => 3%5 => 3
for : rear=3 => rear = (rear+1)%SIZE => (3+1)%5 => 4%5 => 4
for : rear=4 => rear = (rear+1)%SIZE => (4+1)%5 => 5%5 => 0

ii. dynamic implementation of a linear queue (by using linked list)
enqueue : add_last( )
dequeue : delete_first( )

OR

enqueue : add_first( )
dequeue : delete_last( )

- priority queue: it is a type of queue in which, elements can be inserted into it randomly (i.e. without checking priority) from rear end, whereas element which is having highest priority can only be deleted first.

- double ended queue (deque): it is a type of queue in which elements can be added as well as deleted from both the ends.
- there are further 2 subtypes of deque:
1. input restricted deque : it is a type of deque in which elements can be added into it only from one end, wheras elements can be deleted from both the ends.

2. output restricted deque : it is a type of deque in which elements can be added from both the ends, wheras elements can be deleted only from one end.

dfs (depth first search) traversal => stack
bfs (breadth first search) traversal => queue

 # applications of queue:
- queue is used to implement advanced data structure algorithms like **bfs breadth first search traversal** in tree & graph.
- queue is used to implement **kernel (i.e. core program of an OS) data structures** like **job queue, ready queue, message queue** etc.....
- queue is used to implement OS algorithms : cpu scheduling algo's like fcfs scheduling, priority scheduling etc... and disk scheduling algo's like fcfs etc...., and page replacement algo's like fifo, lru
- programs in which, list/collection of elements should works in a fifo manner data structure queue can be used.

 # introduction to an advanced data structure:
+ tree: it is an **advanced / non-linear data structure**, which is a **collection/list of finite number of logically related similar type of data elements** in which, there is a first **specially designated element referred as root element**, and remaining all elements are connected to it in a **hierarchical manner** follows **parent-child relationship.**

- in tree data structure element is also referred as a node.
- root node => first specially designated element in a tree.
- parent node/father
- child node/son
- grand parent/grand father
- grand child/grand son
- ancestors – all the nodes which are in the path from root node to that node
root node is an ancestor of all the nodes

- descedents – all the nodes which can be accessible from it
all the nodes are descedent of root node

- degree of a node = no. of child nodes
- degree of a tree = max degree of any node in a given tree

- leaf node = node which is not having any number child node/s
OR node having degree 0.

- non-leaf node = node which is having any number child node/s
OR node having non-zero degree 0.

- level of a node = level of its parent node + 1
- level of a tree = max level of any node in a given tree => **depth of a tree**

- as tree is a **dynamic data structure:**
  - in tree data structure any node can have any no. of child nodes
  - tree can grow upto any level ...

- operations like addition, deletion, searching etc... on tree becomes inefficient, so restrictions can be applied on it, and hence there different types of tree:

- **binary tree** – it is a type of tree in which each node can have max 2 no. of child nodes, i.e. each node can have either 0 or 1 or 2 no. of child nodes.
OR it is a type of tree in each node can have degree wither 0 or 1 or 2.

- binary tree is set of finite of no. of elements has 3 subsets
1. root node
2. left subtree (may be empty)
3. right subtree (may be empty)

- empty set/null set : set which contains 0 no. of element
- singleton set : set which contains only 1 element

- **binary search tree (bst)** – it is a **binary tree** in which **left child is always smaller than its parent** and **right child is always greater or equal to its parent.**

- traversal on a tree = to visit each node in a tree at max once
- tree traversal can be always starts from root node

- there are 2 basic tree traversal methods in a binary search tree:

1. bfs (breadth first seach) traversal / levelwise traversal:

- traversal starts from root node and each node gets visited levelwise from left to right.

2. dfs (depth first search) traversal :

- there are further 3 types of traversal under dfs:

i. Inorder (L V R)
ii. Preorder (V L R)
iii. Postorder (L R V)

V : Visit
L : Left Subtree
R : Right Subtree

i. Inorder (L V R):

- we can always start traversal from root node
- we can visit left subtree of any node first, then we can visit that node and then we visit its right subtree
- we can visit any node either after visiting its left subtree or its left subtree is empty.
- in this traversal, all the nodes gets visited always in an ascending order

ii. Preorder (V L R):

- we can always start traversal from root node
- we visit the current node, then we visit its left subtree and then only we can visit its right subtree.
- in this traversal, root node always gets visited at first, and this property remains same recursively for each subtree as well.

iii. Postorder (L R V)

- we can always start traversal from root node
- we can visit any node only if its left subtree as well as right subtree are either empty or already visited.
- in this traversal, root node always gets visited at last, and this property remains same recursively for each subtree as well.

- height of a node = max(ht. of its left subtree, ht. of its right subtree) + 1
- height of a tree = max height of any node in a given tree
- max height of bst = n, whereas n = no. of nodes in a bst
- min height of bst = log n, whereas n = no. of nodes in a bst

- left skewed bst : bst in which right link of each node is NULL, each node is having only left child.

- right skewed bst : bst in which left link of each node is NULL, each node is having only right child.

- bst which having max height for given input size is referred as imbalanced bst => as operations like addition, deletion & searching cannot be performed on it efficiently i.e. in O(n) time.

- bst which having min height for given input size is referred as balanced bst => as operations like addition, deletion & searching can be performed on it efficiently i.e. in O(log n) time.

- if bst is not balanced then there is a need to balanced it to achieve effciency in operations on it, we can balance bst by applying left rotations and right rotations as per the requirement.

- if all the nodes in a bst are balanced then only we can say bst is balanced.
- we can node is balanced only if balance factor of a node is either -1 OR 0 OR +1.
- balance factor of a node = ht. of its left subtree – ht. of its right subtree

if balance factor of any node < -1 => node is left imbalanced and hence bst is left imbalanced ==> apply left rotation on it

if balance factor of any node > +1 => node is right imbalanced and hence bst is right imbalanced ==> apply right rotation on it

self balanced bst – it is a bst in which while adding node into it or deleting node from it, it makes sure that bst remains balanced, this concept was designed by two mathematicians : adelson vesinki & lendis
and hence self balanced bst is also referred as avl tree.

+ Data structures : Modular Batch DS

Reference Books:
1. Data Structures using C++ => Sartaj Sahani
2. Design & Analysis of an Algorithms => Coreman
+ graph:
it is an **advanced / non-linear data structure** which is **a collection of logically related similar and disimmilar type of data elements**, which contains:
- finite set of elements called as **vertices** also called as nodes, and
- finite set of **ordered/unordered pairs of vertices** called as an **edges** also called as an arcs, whereas **edges may contains weight/cost/value** and it may be **-ve.**

- whenever there is need to store data in networks => graph

google map:
1000 cities
information about cities => vertices
information about paths between cities => edges

=> collection of vertices and edges => graph

```
typedef struct city
{
        int city_code;
        char city_name[ 32 ];
        .
        .
        .
}city_vert;
```

- information about paths between cities => edges

```
struct path_edge
{
        int src_city_code;
        int dest_city_code;
        float dist_km;
        float time;
        .
        .
        .

}
```

social sites : facebook, linkedin

person => vertex
connctivity between two people => edges

adjancent vertices - if there exists a direct edge between two vertices, otherwise they are referred non-adjancent.

If u, v are adjancent vertices,
if edge between them can be represented either (u, v) OR (v, u) => undirected edge => unordered pair of vertices.

- depends on wheather pairs of vertices i.e. edges are ordered pairs of vertices or unordered pairs of vertices there are two types of graph:
1. undirected graph
2. directed graph (di-graph)

- depends on wheather edges carries weight or not, there are two types of
1. weighted graph : graph in which edges carries weight/cost/value
2. unweighted graph : graph in which edges do not carries weight/cost/value

- loop – if there exists an edge from any vertex to that vertex itself, such edge is referred as a loop.

- path : we can traverse from one vertex to another vertex through path

- if path exists between two vertices then those vertices are referred as connected vertices otherwise they are referred as not-connected.
- adjancent vertices are always connected, but vice-versa is not true

- connected graph – if any vertex is connected to remaining all vertices in a given graph it is referred as a connected graph otherwise it is referred as not-connected/disconnected graph.

- isolated vertex – if any vertex is not adjacent with any other vertex in a graph

- cycle => in a given graph, in a path if starting vertex and end vertex are same, such path is referred as a cycle

G = (V, E)
V = { 0, 1, 2, 3, 4 }
E = { (0,1), (0,2), (0,3), (1,2), (1,4), (2,3), (3,4), (3,3) }

- there are two graph representation methods:
1. adjancency matrix representation (2-d array)

size of the matrix/2-d array => V * V (V = no. of vertices in a graph)
- in a adjancency matrix representation of unweighted graph:
if vertices are adjacent => entry between them = 1
if vertices are not adjacent => entry between them = 0

- in a adjancency matrix representation of weighted graph:
if vertices are adjacent => entry between them = weight
if vertices are not adjacent => entry between them = inf

2. adjancency list representation (array of linked lists)
for each vertex one linked list will be maintained and hence V no. of linked lists will be there into the graph.