DATE 12 06 22 M-4

01. Data Structure And Algorithm using C

* Data structure

Arrangement of the data so that data items can be used in efficient manner.
To acheive efficiency in programming data structures are used. It's the way to store data elements into the main memory in the organized manner so that operations like searching, sorting, traversal, etc can be performed on the data efficiently.

* Algorithm

The sequence of steps followed on the given data using the efficient data structure to solve given problem.

* Database

The collection of information in the storage devices for for the faster retrieval & updation.

*

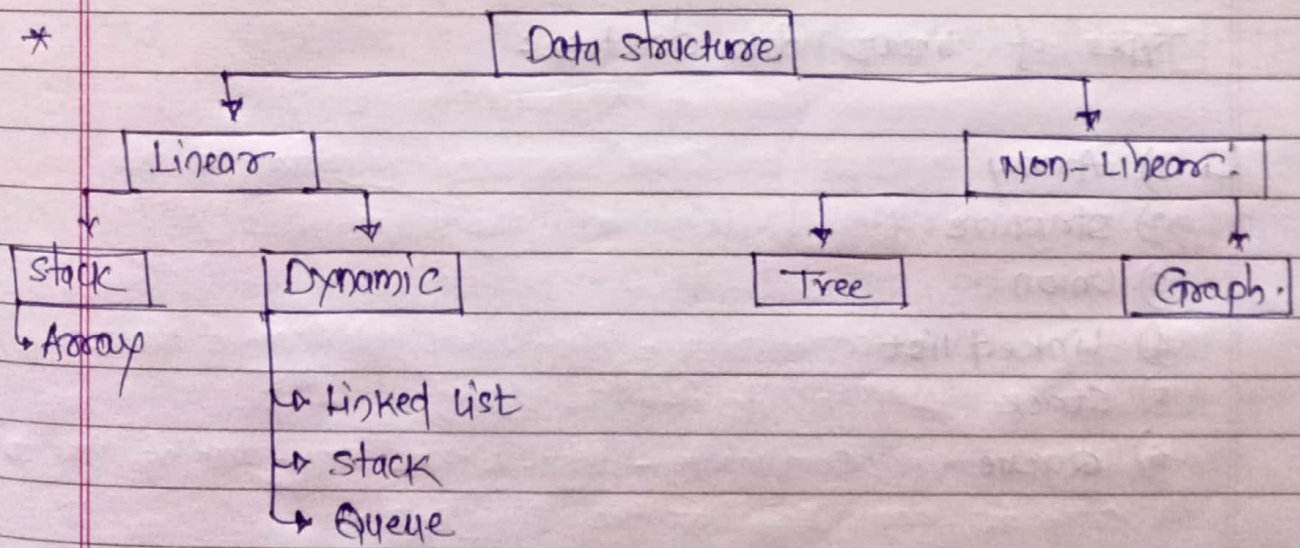


fig- Data structure classification.

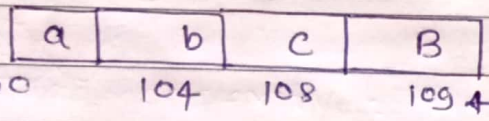
Basically, there are two types of data structure :

01- Linear data structure :

The data structure in which data elements gets stored into the memory in linear manner ie, countigiously hence it can be accessed linearly ie, one after another, so this type of data structure called as linear data structure.

Data arrangement done in linear manners in memory but as per size of the data type.

Ex: int a; int b; int c; char='B';



size of int. data type is 04 bytes that's why a is placed at 100th place but 'b' is placed at 104th place so on.

Types of linear data structure :

1) Array
2) structure
3) Union
4) Linked list
5) Stack
6) Queue

02. Non-linear data structure =

The data structure in which data elements get stored into the memory in hierarchical manner ie, form of non-linear manner so this type of data structure called as Non-linear data structure & they can be accessed in Hierarchical Manner's only.

Types of Non-linear data structure =

Tree.
Graph.
Binary Heap.
Hash table.

*

Program is the finite set of instructions written in progra. language given to the machine to perform specific task.

An algorithm is finite set of instructions if followed. it accomplishes given task.

Program is implementation of algorithm. An algorithm is just like blue print of program on paper.

Pseudocode is raw & basic program with instructions if followed it accomplishes given task.

An algorithm is solution of the problem.
But, one problem may have multiple solution
eg.

Searching & sorting data in correct manner.

If one problem has many solution, need to select efficient solution to utilize the resources in best way & to decide which solution/ algorithm is efficient & for that analysis is done.

Analysis of the program is the work of determining or calculating how much time computer taking & how much memory / space computer needed to run program. So theare are Two Measures :

01] Time complexity :

It's an amount of the time computer needed to run a program for pertiular algorithm.

2] Space Complexity :

It's an amount of space / memory required to run a program for pertiular algorithm.

1.1) Best time complexity : Algo. takes min. amt. of time to run to completion of the program.
1.2) Worst time Complexity - Algo. takes max. amt. of time to run to completion of the program.
1.3) Average time complexity - Algo. takes neither min nor Max ie, avg amt. of time to run completion of program.

Asymptotic Analysis :-

Asymptotic analysis is an mathematical way to calculate time & space complexity of an algorithm without implementing it in prog. language.

Asymptotic notations gives us idea about how good is given algorithm compared to other one.

01. Big Oh (O)

It's used to describe asymptotic upper bound. This notation represents worst time complexity. Mathematically,
If $f(n)$ describes running time of algorithm;
$f(n)$ is $O(g(n))$ if there exists positive constant C & n_0 such that

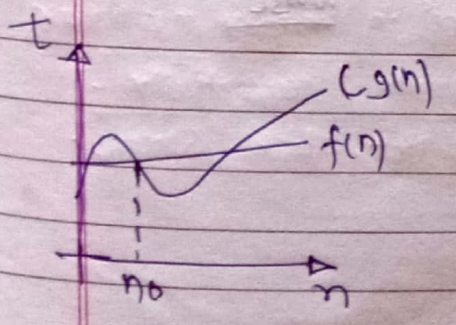
$$0 \leq f(n) \leq g(n) \quad \text{for all } n \geq n_0$$

02. Big Omega (Ω)

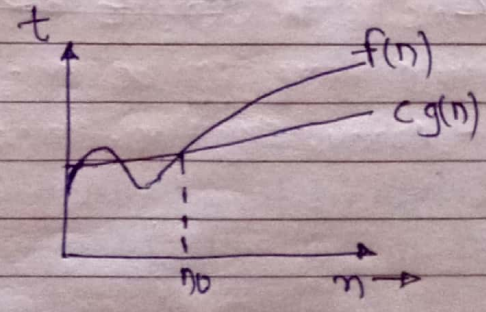
Its used to describe asymptotic lower bound. This notation represents Best time complexity.

$f(n)$ is said to be $\Omega(g(n))$ if there exists positive constant C & n_0 such that,

$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$



— BB Oh —



— BB Omega —

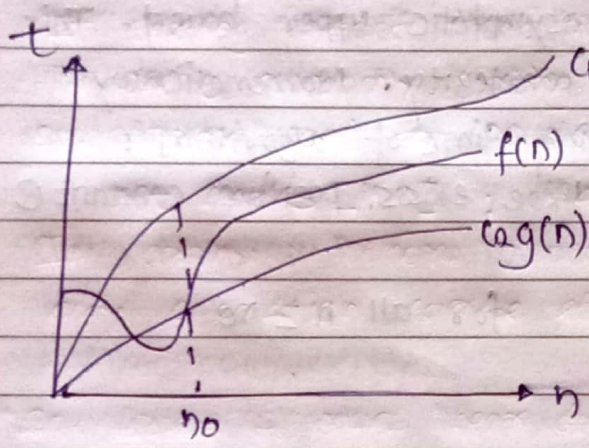
03- Big Theta (Θ)

This notation used to represent average case time complexity. It's asymptotic tight bound.

f(n) is said to be Θ g(n) if f(n) is O g(n) & f(n) is Ω g(n).

Mathematically,

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall n \geq n_0 .$$



— Big Theta —

Increasing order of common runtime

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

↑ Better — ↑ common runtime — ↑ Worst

Linear search / sequential search

In this algorithm, key elements gets compared with each array element by traversing it from first element till the either match is found or max. till the last element.

* Algorithm Linear Search (A, size, key).

```
{
for (index = 1 ; index <= size ; index ++)
 { if (key == A[index])
    return true;
 }
 return false;
}
```

arr	10	20	30	40	50	60	70	80
	0	1	2	3	4	5	6	7

* If key is found at very first position of comparison then it is called as best case $O(1) = \Omega(1)$

* If key is either found at last position or key doesn't exists, for which n no. of comparisons takes place then its worst case & running time is $O(n) = O(n)$.

* If key is found b/w in any position in b/w array then its avg. case & running time is $O(n/2) = \Theta(n)$

* Not everyone's computer is equally powerful; asymptotic notation is measure of how time (runtime) grows w.r.t. input so that complexity can't be calculated in seconds.

Data Structure And Algorithms.

Day 01

✓ Data structure is programming concept used in every language. It's way to store data elements into the main memory ie RAM in organised manner so that operations can be performed on it efficiently.
Data is always processed in main memory.

✓ To store data of 100 students, 100 variable created & takes memory of 400 bytes.

```
int m1, m2, m3 -- m100; // 400Bytes
```

If sorting needed then; these variable stored in memory anywhere randomly, sorting operation will be difficult.

✓ So, Array used to 100 student's mark. Then size of array

```
int marks[100] ; //400 bytes.
```

In array, memory allocation in countigous manner so any operation can be efficiently used/performed.

✓ So, Array is basic linear data structure, which is collection/list of logically related similar type of elements stored in countigous manner.
That's why Array is datastructure.

✓ There are two types of data structures:

01) Linear data structure.
02) Non-linear data structure.

✓ 01) Linear / Basic data structure.
In this, data elements can be stored in memory in a linear manner hence can be accessed linearly / sequentially.
- Array, Structure, Unions, linked list, Stack, Queue.

02) Non-linear / Advanced data structures.
In this, data elements gets stored into memory in non-linear manner and hence can be accessed non-linearly.
- Tree (heirachical manners)
- Graph, hast table, binary heap

If different data type of element stored then structure used.
structure is basic linear data structure, which is collection of list of logically related similar & dissimilar type of elements gets stored into memory collectively as single entity.
structure is also data structure.

Program is set of instructions written in programming language given to m/c to do specific task.
Algorithm is finite set of instructions written in human understandable language with some programming constraints if followed accomplish given task. This is also known as pseudocode.
flowchart diagramatically represents algorithm - (CCAT mcq - 1st slot)

Traversal: Basic operation applied on data structure in which each data element gets visited at once.
Traversal on array is also called as Array Scanning.

One line algorithm is an solution of given problem.
Sorting: It's an arranging data elements in collection or list of element's either in ascending or descending order. By default sorting is in ascending order.
There are diff. sorting algorithms -
1. selection sort
2. Insertion sort
3. Bubble sort
4. Quick sort
5. Merge sort
6. radix sort
7. shell sort

One problem may has many solutions, then efficient solution prefer

"Searching"

To search a particularly key element in collection / list of elements is searching. It has linear search, binary search.

✓ When efficient solution preferred, analysis of solutions required for that algorithm.
Work of determining computer time & computer memory required for that algorithm to run to completion.
To calculate Time Complexity & Space Complexity are two measures of analysis of algorithm.

✓ Time Complexity: Amount of time required for it to run to completion.

✓ Space Complexity: Amount of space required for it to run to completion.

✓ Linear Search Algorithm:

In this, key elements i.e, searching element gets compared sequentially with array element by traversing it from first element till either match is found or till the last element.

Eg:

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

index

Key = 30 . (To search)

✓ The key = 30 if found at very first position in array only in one comparison. key found in one comp. is Best case. And Running time = O(1) ∴ (as no. of comparison is only one 1). If key=30 occured twice in array, only first case will considered.

✓ key = 10 (To search)

key = 10 found at last position in array with 10 comparisons. i.e. All possible comparisons called Worst case (even key can't found)

∴ Running time = O(n) where n is size of array.

✓ key = 100

In this case algorithm neither takes min or max so this called as Average case.

988	30	20	90	60	80	70	100	50	40	10
	0	1	2	3	4	5	6	7	8	9 ← Index

Time Complexity

~~SS~~ key	case	comparisons	parameters	Running Time
~~O(1)~~ 30	Best	01	key found at first position	O(1)
O(n) 10	Worst	10	key found at last position	O(n)
O(n) 55	Worst	10	key doesn't exists	O(n)
O(n/2) 100	Average	07	key found in b/w first & last posit	O(n/2)

01) Best case time complexity – Key Algo. takes min. amount of time to run completion

02) Worst case time complexity – Algo takes max. amount of time to run to completion

03) Average case time complexity – Algo takes neither min nor maximum amt. of time to run to completion.

✓ Analysis / math. way to calculate Time & space complexity without implementing it in any programming lanuage called as Asymptotic Analysis.

In asymptotic analysis, analysis is done depending upon basic operations in that algorithm.

eg- Searching / sorting : Basic operation "comparison"

✓ Asymptotic Notations :

Rules & certain notations needs to be followed.

01) Big Omega : Ω / $-\Omega$

This notation is used denote best time complexity of an algorithm.

02) Big Oh

This notation is used to denote worst time complexity.

03) Big Thetz (Θ)

✓ This notation is used to denote avg. time complexity. If running time of any algorithm has additive/ subtrw/div. constant then it can be neglected. (discrete math. Rule)

∴ Avg. complexity (time) = $\Theta(n/2) \cong \Theta(n)$.

✓ Avg. Time complexity $\cong \Theta(n)$

1.24-30

Binary search :

To apply binary search, array must be sorted, If array is not sorted, binary search can't be applied.

This algorithm follows divide & conquer approach.

eg-

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

mid (1st)

$mid = \frac{left + right}{2} = \frac{0+9}{2}$

mid = 4

Left Sub Array (LSA)

Right Sub Array (RSA)

10	20	30	40
0	1	2	3

Left Right

Key compared with mid & accordingly if key is greater or lower further iterations followed.

60	70	80	90	100
5	6	7	8	9

L mid (2nd) R

$mid = \frac{5+9}{2}$

mid = 7

60	70
5	6

90	100
8	9

L mid (3rd) R

LSA Invalid (No array Avai)

100
9

R/L (4th)

Key = 50 - 1st Iteration

Key = 90 - 3rd Iteration

Key = 100 - 4th Iteration

key	case	No. of compar.	Parameters	Running Time	Time Complexity
50	Best	1	key found at 1st iter.	O(1)	$\Omega(1)$
90	Avg	log(n)	key found in b/w 1st & last it.	O(log(n))	$\Theta(\log n)$
100	Worst	log(n)	key found at last iter.	O(log(n))	$O(\log n)$

Whenever divide & conquer approach → 'log n' used.

Procedure -

Sorted Array given, then mid calculated using (left+Right)/2. key is compared with mid. If not matched then key value compared greater / lower way of iteration proceeds until key found. Mid Always in int value

Algo - with divide & conq. approach has Time Complexity in terms of log n

* When two algorithms compared w.r.t. their efficiency their Average case Time complexity should be considered.

* log n is efficient than n.

Data structures = Sorting Algorithms

01. Selection sort.

- In this algorithm in first iteration, first position gets selected & element which is at selected position gets compared with all its next position elements.
- If selected position element found greater than any other position element then swapping takes place & in first iteration the smallest elements gets settled at first position. Similarly for 2nd iteration till in maximum (n-1) no. of iterations all array elements gets arranged, in sorted manner.

By default, sort it in ascending order.

To sort all element total iteration required = n-1

n is size of array.

In selection sort - iteration-1 has (n-1) comparisions

As per rule additive/sub/mul/div. constants should be neglected.

$\therefore O(n-1) \rightarrow O(n)$.

$\therefore$ Iteration $(n-1) \rightarrow O(n)$

$\therefore$ Total no. of comparisons $= n(n-1) = \underline{n^2-n}$

Rule-2 Hence if running time of any algo. has polynomial in its time complexity only leading term considered

$\therefore O(n^2-n) \Rightarrow \boxed{O(n^2)}$

	Best Case	Worst Case	Avg Case	Time Complexity
Selection sort	$\Omega(n^2)$	$O(n^2)$	$\Theta(n^2)$	

selection sort eg -

Iteration - 1

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

swapped as sel pos < pos

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

20	30	60	50	10	40
sel pos			pos		

20	30	60	50	10	40
sel pos				pos	

10	30	60	50	20	40
sel pos					pos

Iteration - 2

10	30	60	50	20	40
	sel	pos			

10	20	60	50	20	40
	sel		pos		

10	30	60	50	20	40
	sel			pos	

10	20	60	50	30	40
	sel				pos

10	30	60	50	20	40
sel pos					pos

And iteration continues till sorting

final sorting = | 10 | 20 | 30 | 40 | 50 | 60 |

Why time complexity is same for all cases ?

Even though all elements are in sorted manner like 10 20 30 40 50 60 it will perform all iterations - 5 - hence, running time is same (n^2) for all cases -

In selection sort, smallest no. is settled first & then last no -

02. Bubble sort / sinking sort

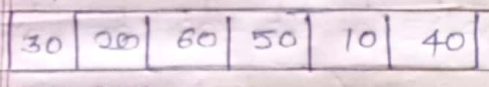
In bubble sort two consecutive places of array compared first. In this largest element is settled at last position first. In first iteration 2nd largest element is settled at 2nd last position & so on.

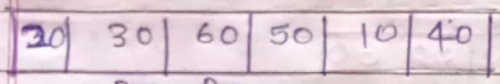
Total no. of iterations takes place (n-1) if array is not sorted.

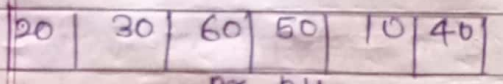
Efficiency of buble sort is more efficient than selection sort only in implementation order.

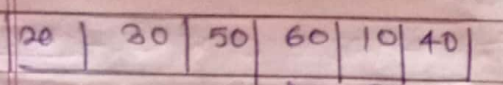
But analysis wise - selection & bubble sort is same.

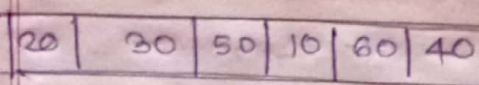
Iteration-1

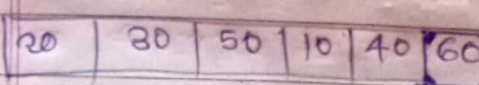


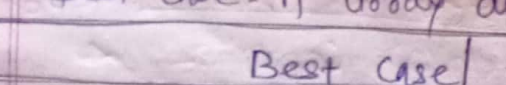




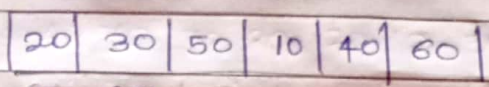


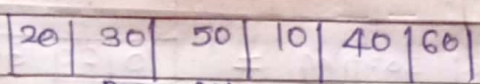


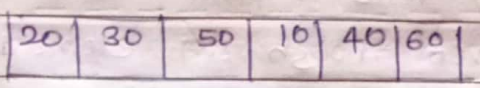


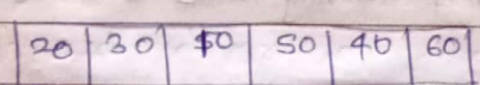


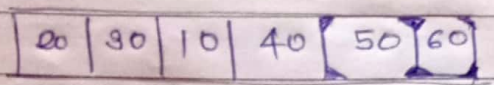
Iteration-2











And iteration 3,4,5 countinues.

* Best case: If array already sorted → O(n)

	Best case	Worst Case	Avg case
Bubble sort	Ω(n)	O(n²)	Θ(n²)

03 Insertion Sort

It works like sorting of playing cards in left hand (realtime eg). Sort elements of array by inserting key in every iteration at its appropriate position. In this also, unnecessary comparisons avoided. All elements of array set shifted to left side by one place.

In this, in every iteration, one element is picked from an array which is refered as key element & compare element with left hand side from Right to left & whenever appropriate position it will be inserted.

Insertion sort algo. is efficient for already sorted array. Also it is efficient for small data values. Insertion sort is adaptive in nature.

* working of insertion sort algorithm

arr[] = {12, 11, 13, 5, 6}

Iter-1

12	11	13	5	6

Initially, first two elements 12, 11 compared. Here, 12 > 11. Thus swap happens

Iter-2

11	12	13	5	6

Now, 12 & 13 compared but 12 > 13 No swap.

Iter-3

11	12	13	5	6

Next, 13 < 5 so 5 will be at 13th place but 12 < 5 & 11 < 5 so 5 is settled at left

Iter-4

5	11	12	13	6

Now 13 < 6 thus 6 is compared with 13, 12 & 11 < 6 but 6 is > 5 so 6 will be placed

5	6	11	12	13

// To sort array of N size in ascending order

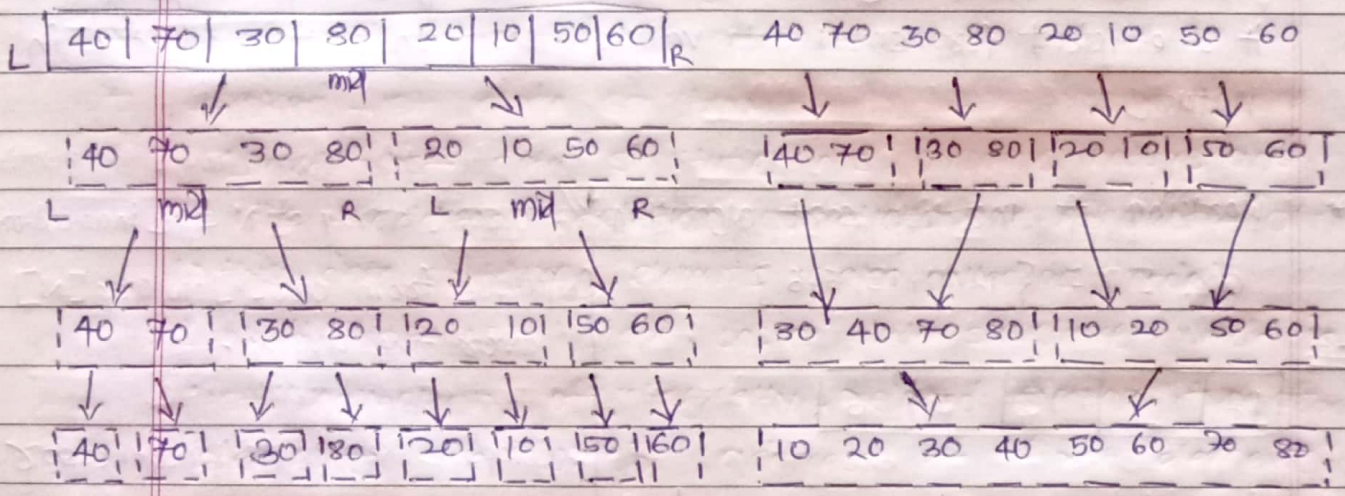
- Iterate from arr[1] to arr[N] over the array.
- Compare key to its predecessor
- If key is small than predecessor compare to element before. Move greater element one position up to make space for the swapped element

Best	worst	Avg
$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$

04. Merge Sort algorithm

This algorithm based on divide and conquer strategy. So array is initially divided into two equal halves & then they are combined in sorted manner. If the array becomes empty or has only one element left, dividing will stop.
Finally, when both halves sorted, then merge operation is applied (Merge: Process of taking two smaller sorted arrays & combining them to eventually make larger one.)

eg:-



step01 - Declare Array & left, right and mid variable
step02 perform merge function
step03 - Array sorted -

↳ Merge sort is slower comparatively to other sort algorithms for small tasks
↳ It requires additional memory space for temp. array
↳ Even if array is sorted, it goes through whole process.

Merge sort	Best	Worst	Average
↳	Ω(nlogn)	O(nlogn)	Θ(nlogn)

* Time Complexity -

Algorithm	Best Case	Worst Case	Average Case
Selection sort	$\Omega(n^2)$	$O(n^2)$	$\Theta(n^2)$
Bubble sort	$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$
Insertion sort	$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$
Merge sort	$\Omega(n \log n)$	$O(n \log n)$	$\Theta(n \log n)$
Quick sort	$\Omega(n \log n)$	$O(n^2)$	$\Theta(n \log n)$

Day 02.

* Limitations of Array datastructure :-
  - Array is collection of similar elements
  - Array is static
  - size of array can't be grow/shrink during runtime.
  - Addition & deletion operation are not efficient.
    eg. int arr[1000]; → Add element at 485th place then all elements's needs to shifted at Right side.
    eg. int arr[1000]; And store 995 elements only then 20 Bytes wasted

Hence, to avoid / overcome above limitations linked list Data structure has been designed.

* Linked list data structure -
  Linked list data structure has dynamic nature and it's addition, deletion operations are efficient - O(1) time.
  In this data structure, "elements" is also called as "node"
  Linked list is linear, basic data structure, which is list / collection of logically related similar type of element in which address of first element in it gets stored into the pointer variable referred as "head", each element contains actual data and an address of its next element (as well as address of its previous element.)

  Types of link list :
  01. Singly linked list.
  02. Doubly linked list.

01) Sinsly linked list : In this, each element/node contains an address of its next node. It's further divided into :
  - Sinsly linear linked list
  - Sinsly circular linked list

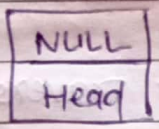
02) Doubly linked list - In this, each element/ node contains address of its next node as well as address of its previous node. Doubly linked list further divided into
- Doubly linear link list.
- Doubly circular link list.

✓ Basically all these 04 types are evolved to overcome limitation of its previous link list.

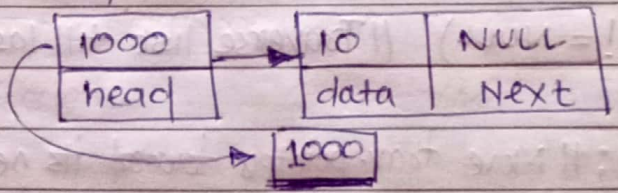
01) Singly linear linked list: In this data structure
1. head always contains address of first node/element if list is not empty.
2. each node has two parts :-
   - "Data": it contains actual data of primitive/nonpri.
   - "pointer" (next): Contains address of its next node.
3. Last node points to NULL ie, next part of last node contains NULL.

✓ Traversal is very important in linked list

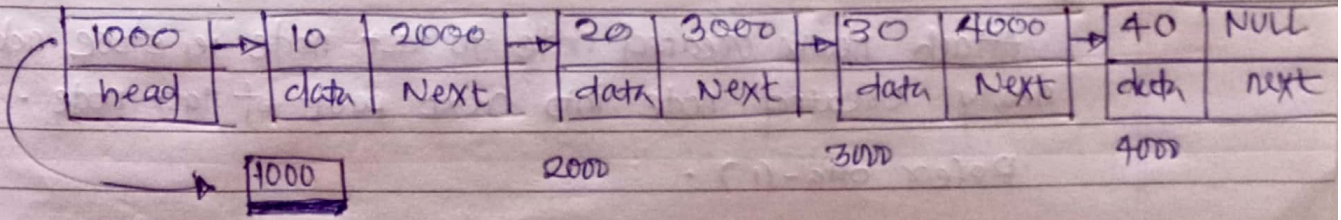
01) singly linear link list (When list is empty)



02) singly linear linked list (When list contains only one node)



03) Singly linears linked list (When list contains more than one node)



✓ Addition of node into the list

- Add node into the list at last position
- Add node into the list at first position
- Add node into the list at specific position

✓ Deletion of node into the list

- Delete node from list which is at first position
- Delete node from list which is at last position
- Delete node from list which is at specific position

* Add node into the list at last position:

Step 01 - Create a newnode by using dynamic memory allocation

Step 02 -

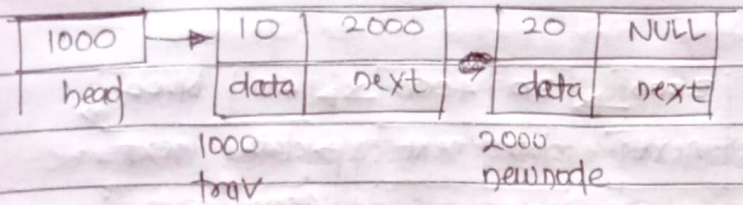
if the list is empty then attach newly created node to the head

```
if ( head == NULL) {
head = newnode ; // store add. of newly created node into head.
   }
else  // if node is not empty, start traversal from
          first node.
 trav = head ; // Add. of first node can fetched from head pointer
while ( trav -> next != NULL) // Traverse list till last node
{
trav = trav -> next ; // Move trav. pointer toward its next node
}
  trav -> next = newnode ;
}
```

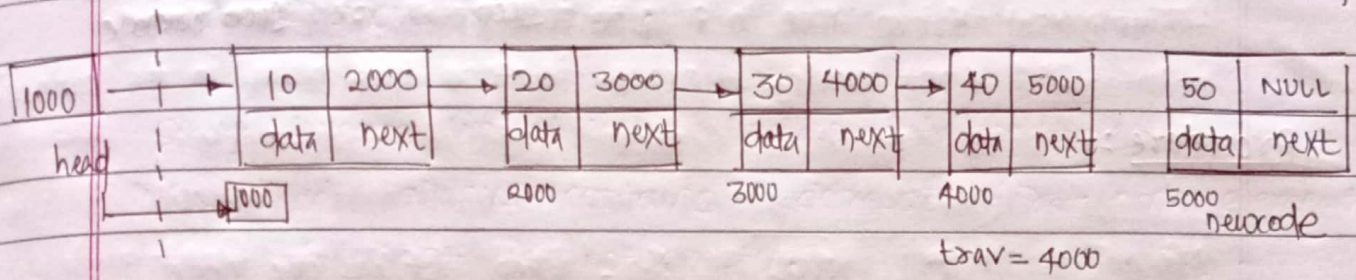
Attach newly created node to last node ie, store addr of newly created node into next part of last node.

Refer dia-(1).

add node into singly linear link list at last position : if list is empty -

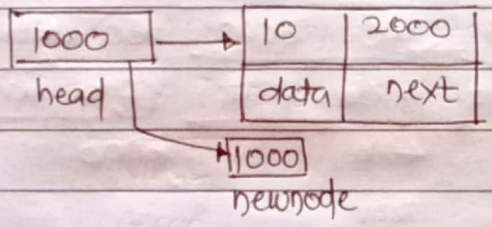


Add node into singly linear link list at last position : if list is not empty -



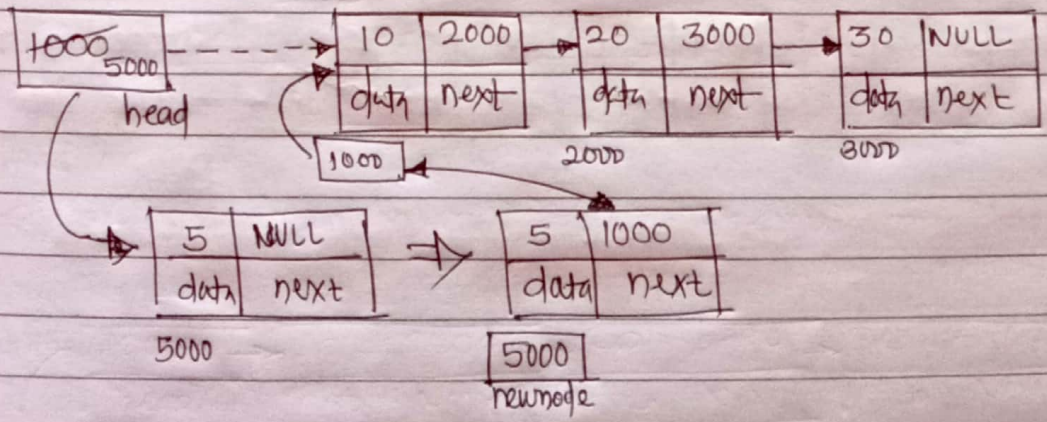
* 'n' no. of nodes can be added in O(n) time is SLLL -

Add node into SLLL at first position : if list is empty -



* 'n' no. of nodes can be added in O(1) time

✓ Add node into SLLL at first position : if list is not empty -



At first head = 1000 but newnode is placed at (5000) head & new connection formed -

Deletion process

Delete node from SLLL at first position

step01 - If list is empty print "list is empty"

step02 - If list is not empty then delete node from it at 1st position

```
if (head != NULL)
{   // if list contains only one node
    if (head -> next == NULL)
    { free (head);  // delete node & make head as NULL.
      head = NULL;
    }
  else // if node list contains more than one node
  { temp = head; // store add. of current first node into temp.
    head = head -> next; // store add of cur. second node into head
    free = temp; // delete the node
  }
}
```

Delete nodes from SLLL in O(1) time.

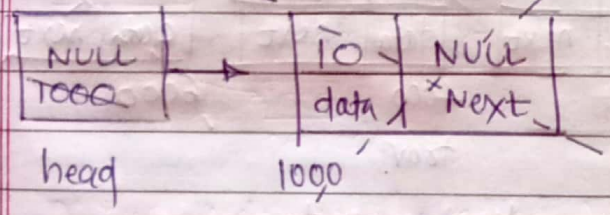
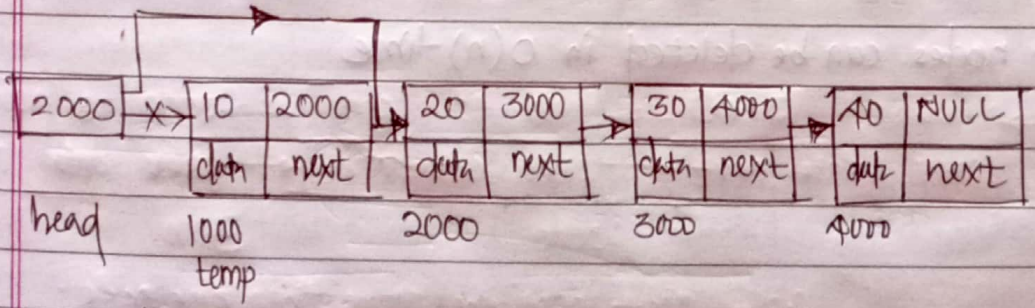


fig - if list contains only one node



head = head -> next

fig - link list contains more than one node.

Delete node at last position

In this case, singly linked list allows only forward traversal, previous node can't be accessed.
To delete node at last position need to traverse from first node to 2nd last node.
Traversal from 1000 to 2000 to 3000 uptill 2nd last node.
In 2nd last node's next ie, 6000 has NULL in its last node.

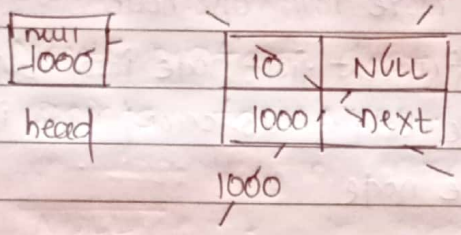
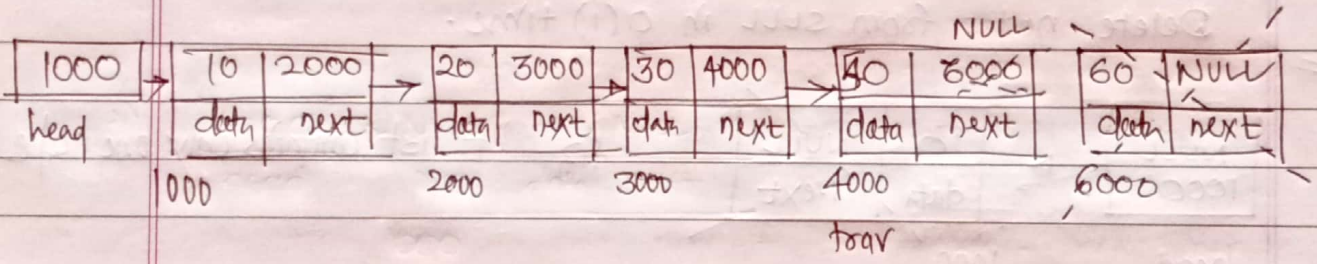


fig- when list has one node



- fig- when llst has onore than one node-

No. of nodes can be deleted in O(n) time

Limitation's of Singly Linear Link List

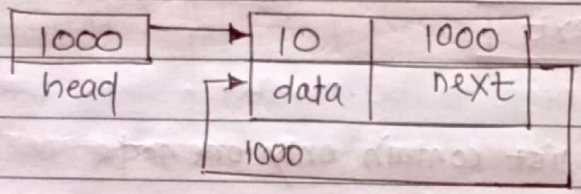
01) Traversal allowed only in one direction
02) Previous node can not be visited.
03) Previous node can't be accessed from any node of it
04) Adding & deleting last operation inefficient takes O(n) time.

To overcome this limitation singly circular link list designed.

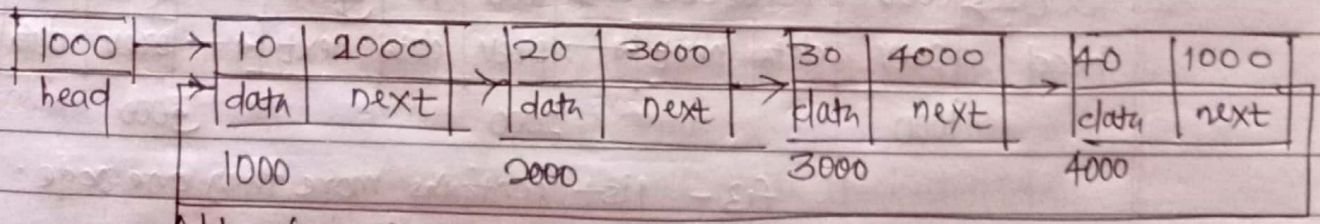
* Singly Circular link list ::>

1) NULL
   Head

2) Singly Circular link list -> list contains only one node.



3) Singly Circular link list -> list contains more than one node



Add. of next last == Add. of head.

✓ Adv - Revisit any node

✓ dis - 1) Due to traversal of every node, add last, delete last, add first, delete first are not efficient as it will take O(n) time.
2) Traversal in forward direction - 3) Can't access prev node.

To overcome limitations of singly circular & singly linear link list doubly link list is designed.

Doubly linear link list :

In this, head always contains address of first element if list is not empty.

Here, each node has Three parts

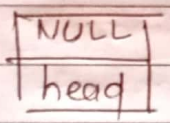
i) data : Contains data of pri/nonprimitive type.

ii) pointer part (next) : contains add. of next node.

iii) pointer part (prev) = Contains add. of prev. node.

Next part of last node & prev. part of first node points to NULL.

1)



2)

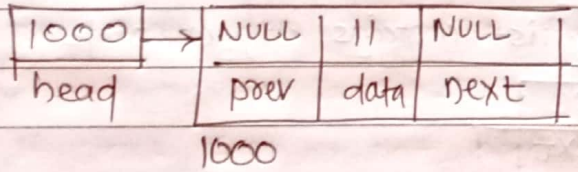


fig- list contain only one node

3)

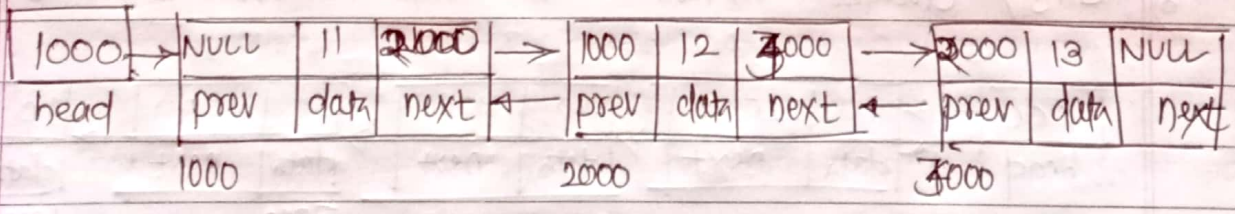


fig- list contains more than one node.

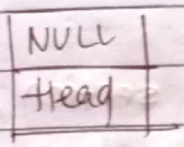
Limitations of Doubly linear link list

↳ Add last & delete last operations are not efficient as it take O(n) time
↳ Traversal only possible from first node

Doubly Circular Link List

In this head always contains address of first node if list is not empty.
* In this list, next part of last node contains address of first node & previous part of first node contains add of last node.

01)



list is empty

02)

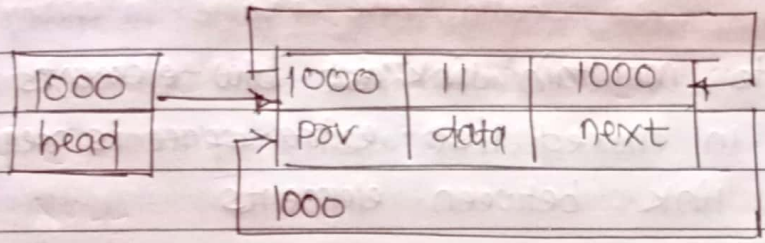


fig- list contains only one node

03)

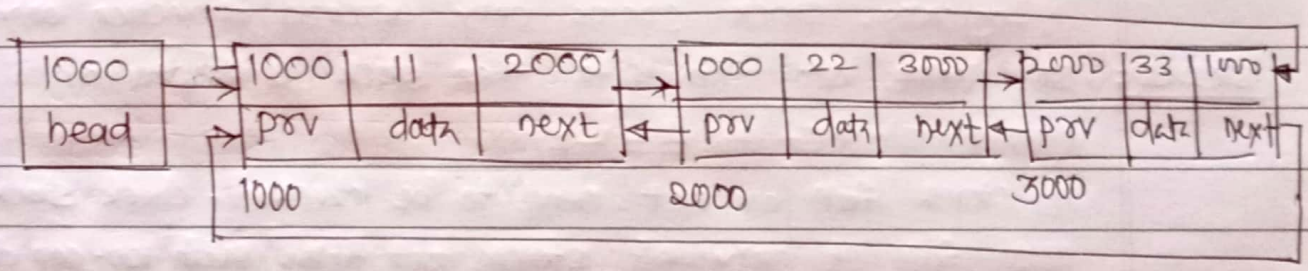


fig- list contains more than one node

* Advantages -

1. Traversal in forward / Reversal direction possible.
2. Add last, Add fir, delete last, delete first operations efficients as it takes only O(1) time.
3. Traversal possible from first node & last node
4. Any node can be revisited
5. So, any prev. node of node can be accessed -

Array Vs linked list

Array is static ; linked list dynamic

Array elements can be accessed by random access method which is efficient than linked list which accessed by sequential access method.

Addition & Deletion operations are efficient on linked list than on array.

Array elements get stored into stack section, whereas linked list elements gets stored into heap section

In Array, to maintain link ~~list~~ blw elements is done by compiler, in linked list extra space is required to maintain link between elements

stack :-

It's linear basic data structure, which is collection of / list of logically related similar type of elements into which elements can be added as well as deleted from only one end referred as "top" end.

In this, element which was inserted / added last can only be deleted first so called as LIFO ie Last In First Out or FILO ie First In Last Out.

Logically stack is dynamic data structure.

Three basic operations performed onto stack in O(1) time

01) Push - Add / insert element onto stack at top end
02) Pop - to delete / remove element from stack which is at top.
03) Peek - To get value of topmost element without push / pop - Peek is operation.

Traversal, insertion, etc operations are not allowed in this stack.

Stack can be implemented by using two ways
↳ static stack (by using Array)
↳ Dynamic stack (by using linked list)

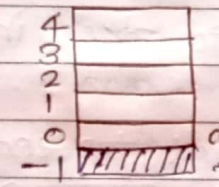
Day 03.

✓ Static stack using array :

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

arr : int [ ] :- Non primitive datatype
top : int :- primitive datatype.

Static stack:



Top should have -1 value. In array indexing start from zero. If top=0 started, then it idicates that stack contains 1 elem

* Push operation :

Logically, stack grows only upward direction only.

Step 1 : So value of top is -1 then stack is empty.
ie, stackEmpty == -1
If stack full or not, need to be checked. check stack is not full & only if stack not full, push op. can done.
Element can not be pushed at -1 value as array indexing starts only from 0.

Step 2 : Thus, value of top needed to increament very first.

Step 3 : Insert element onto stack at top position. Push element 11 at off = 1th position. ie, 1th position is top position now.

Step 4 : Now, insert one element more, repeat step # 1 & 2 again. And value of top incremented by 1.

When value of top becomes (array size - 1) then stack is full.

∴ Top == (size - 1)

ie, Last first in will be last out.

✓ Pop operation:

If pop op. required, then first check stack is empty or not. Only from Top end, pop can be done. Thus decreament value of top by 1 ie, deletion of element from stack. So, whichever is above top, element is not be in stack. Here, Last in first out in this order pop is acheived. All of operation done in O(1) time.

✓ Peek Operation

To get value, at top most position check stack is not empty. Now, get value of topmost element without increasing or decreamenting value of top.

If peek operation performed one more time for that pop op. needed for that by mean prev. pop element deleted.

POP / Peek allowed only if stack is not empty.

Dynamic stack (By using linked list)

Push : Add_last ( )
Pop : delete_last ( )

head -> 10, 20, 30, 40, 50 (adding last)
head -> 10, 20, 30, 40 (delete last)

Thus, it's in Last In First out way.

OR//
Push : add_first ( )
POP : delete_first ( )

head -> 30 -> 20 -> 10.
Thus, it works in Last in First Out.

Applications of stack :

01. Used by OS to control flow of execution of programs.
02. In recursion, internally OS uses stack.
03. Undo, Redo operations are also implemented using stack.
04. Stack is used to implement advanced data structure like trees, etc.
05. Stack used in to convert given infix expressions into its postfix and prefix and for postfix expression evaluation.

Expression is combination of operands & operators.
There are three types of expressions:

01) Infix expression: Operator is in b/w operands (a+b)
02) prefix expression: Operator is before operands (+ab)
03) postfix expression: after operands (ab+)

01. Infix expression:

priority [*, /, %] > priority [+, -]
when equation same same priority then solve from left to Right

eg- Convert infix eqn into postfix: a*b*c/d*e+f*g-h

⇒ Here, a*b*c/d*e+f*g-h:-

1) ab**c/d*e+f*g-h
2) ab*c*/d*e+f*g-h
3) ab*c*d/*e+f*g-h
4) ab*c*d/e*+f*g-h
5) ab*c*d/e*+fg*-h
6) ab*c*d/e*fg*+-h
7) ab*c*d/e*fg*+h- (equi. postfix express

eg: Convert infix to prefix: a*b*c/d*e+f*g-h

Here: a*b*c/d*e+f*g-h.
↳ *ab*c/d*e+f*g-h
↳ **abc/d*e+f*g-h
↳ /**abcd*e+f*g-h
↳ */**abcde+f*g-h
↳ */**abcde+*fg-h
↳ +*/**abcde*fg-h
↳ -+*/**abcde*fgh (equiv. prefix exp.)

Paranthesis highest priority.

Infix expression : $((a+b) * (c-d)) * e+f * g-h$

↳ postfix expression : ?

↳ $((a+b) * (c-d)) * e+f * g-h$

↳ $(ab+ * cd-) * e+f * g-h$.

↳ $ab+cd-* * e+f * g-h$

↳ $ab+cd-*e* + f * g-h$

↳ $ab+cd-*e* + fg* -h$.

↳ $ab+cd-*e* fg* + -h$

↳ $ab+cd-*e* fg* +h-$

✓* Algorithm to convert given infix into its equi. postfix.

input : $a*b*c/d *e +f *g-h$.

O/p :

stack : (operator) : Check stack is empty or not.

step 1: start scanning infix expression from left to right.

step 2: if (current element is operand)
append it into postfix expression
else if (curnt element is an operator)
{ while (! isstackempty && priority (topelem) >= priority (currentele))
{
pop element from stack & append it to postfix exp.
}
push current element on stack
}

Step 3: Repeat all steps till end of infix expression

step 4: POP all elements from stack one by one & append them onto stack.

* Algorithm to convert given infix into its equivalent prefix -

input : a * b * c / d * e + f * g - h

step1 - start scanning infix expression from Right to Left.

step02 :

```
if (current element is an operand)
   append it into prefix expression.
else // if current element is an operator
   {
      while ( ! isstackempty && priority (topelem) > priority (currentelem)
      {
       pop element from stack & append it into prefix expression
      }
      push current element onto stack
   }
```

step03 : Repeat all above steps till end of infix expression

step04 : Reverse prefix expression -

o/p : **- + * / * * abcde * fgh** ✓

* **Postfix Evaluation**

Convert infix expression into postfix expression : 4*5+8-7/3+4

↳ 4*5+8-7/3+4

↳ 45*+8-7/3+4

↳ 45*+8-73/+4

↳ 45*8+-73/+4

↳ 45*8+73/-+4

↳ 45*8+73/-4+

* Queue :

Queue is basic linear data structure which is collection / list of logically related similar type of element in which, element can added into it from one end ie rear end and element can be deleted from it another end ie frontend.

In this, list element which was inserted first gets deleted first ie LILO or FIFO list.

On Queue data structure, two operations performed in O(1) time
1) enqueue -
2) dequeue

✓ enqueue is to insert / add / push an element into queue from rear end.

✓ dequeue is to delete / remove / pop an element from queue which is at front end.

✓ There are different types of Queue:
- Linear Queue (working type FIFO)
- Circular Queue (working type FIFO)
- Priority Queue -
  It is type of queue in which elements can be added into it without checking priority, from rear end whereas element having highest priority gets deleted first. Each element carries priority with it.
- Double Ended Queue "deque" :
  In this queue, in which elements can be added or deleted from both the ends ie rear & front end.

There are two types of deque
↳ input restricted deque ↳ o/p restricted deque.

I/p restricted deque: In this, elements can be added from only one end whereas, elements can be deleted from both ends.

O/p restricted deque: In this, elements can be added from both ends & elements can be deleted from one end.

* Linear Queue:

01) Enqueue:

step 01: Check queue is not full.
step 2: Then value of rear should increament by 1. And Insert element into queue from rear end.
step 3: if (front == -1) then front = 0.

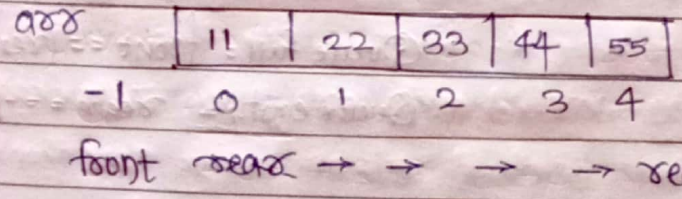


fig- Enque Operation

step 4: If new element needs to be added follow step 1 & 2 & 3.
step 5: If value of rear == (size-1) then queue is full.

02) Dequeue :

Step1: Check queue is empty or not? If queue is not empty then only dequeue possible.

Step2: Increament value of front by 1 ie deleting an element (poping) from queue.

When (Rear = -1 || front > rear) then queue is empty.

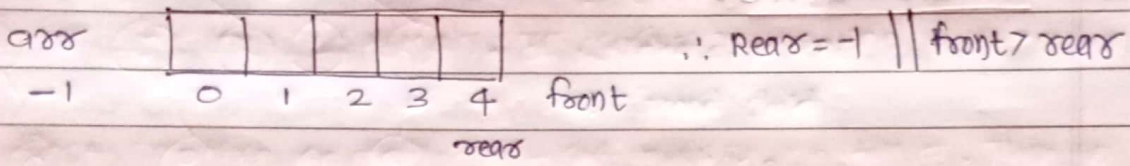


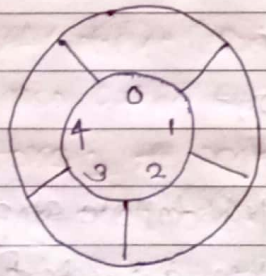
fig- dequeue operation

In linear queue vaccant places can not be reutilized. For this, to overcome this limitation circular queue has been desgn.

* Circular Queue :-

Queue full = front == (rear+1)%size

Queue empty = rear == -1



01) Enqueue -

Check queue is not full
value of rear incremented by +1
Insert / push an element into queue at rear position
if (front == -1), then front = 0;

02) Dequeue -

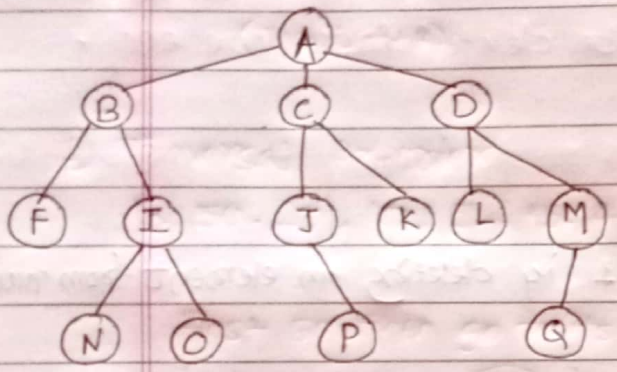
Check queue is not empty
Increment value of front by 1 ie deleting an element from queue
if (front == rear)
front = rear = -1 ; // re-initialize Circular Queue
else,
Increament value of front by 1 (ie deleting element from queue)
[front = (front + 1) % size]

Applications of Queue:

- Job Queue, ready Queue, waiting Queue, etc
- FCFS CPU scheduling, Priority CPU scheduling, page replacement, etc
- Used in BFS (Breadth First Search) Traversal in tree & graph -
  BFS - Queue.
  DFS - Stack.

* Tree:

Tree is the advanced data structure, non-linear in nature, which is collection / list of logically related finite no. of elements, in which there is first specially designed elements referred as "root" element & rest all elements connected to root element in hierachical manner, follows parent - child relationship.



A : Root element/node.
: Parent node
: Child node
: Grand parent
: Grand child
: Siblings

Fig: Tree Data Structure.

- Degree of Node: No. of child nodes having that node.
  Here, (A) has (B), (C), (D) as child so degree of node (A) = 3

- Degree of Tree: Maximum degree of any node in given tree.
  Node (A) having 3 child nodes ie, max called deg. of tree = 3.

- Leaf node: Node having no child node or having zero degree.
- Non-leaf node: Node having child node or having more than 0 degree.
- It's also called as non-terminal or internal node.
- Ancestors: All nodes which are in path from root node to that node including root node, Here, Ancestors of O: A, B, I.
- Descendents: All nodes which can be accessible from that node
  (J), (K), (P) are descendents of (C).
  All the nodes are descendents of root node.

Level of node: level of it's parent node + 1.

Level of tree: Max. level of any node in given tree

Depth of tree: = level of tree.

In tree, any node can have any no. of child nodes & it can grow at any level.

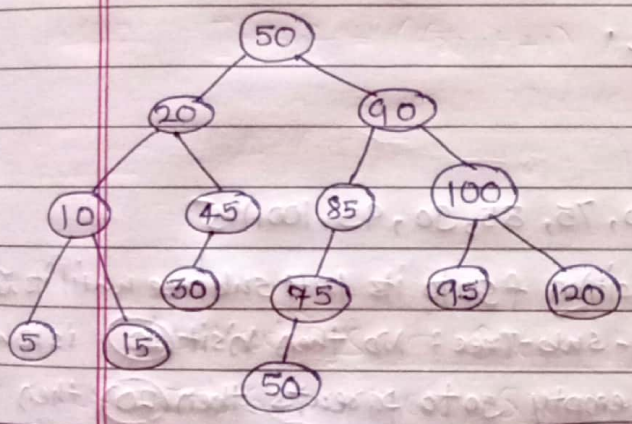
↳ Types of Tree:

↳ Binary Tree: In this, each node can have max. two no's of child nodes. It has degree either 0, 1, 2. Binary Tree has 03 subsets:

01. Root node
02. Left subtree (may empty)
03. Right subtree (may empty)

(BST) ↳ Binary Search Tree: In this, left child is always smaller than its parent & right child always greater than or equals to its parents.

I/p order of an elem's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50



First 50 is root node, then 20 is less than 50 i.e. it will be on left side. 90 > 50 so it will be right side. 85 > 50 & < 90 so it will be left of (85). 10 < 50 & < 20 so it will be left side of 20. 45 < 50 & > 20 so it will be on right side of 20 so this way, BST formed.

DFS (Depth First Search Traversal):

↳ Pick root node first, then go to its depth of left subtree until any childs become NULL till visit i.e. 50, 20, 10, 5 Now depth over. Now 15. then 45, 30. Now its depth completed. Now go to right 90, 85, 75, 50, Now to 100, 95, 120.

↳ I/p = 50, 20, 90, 85, 10, 45, 30, 100, 15, 75, 95, 120, 5, 50

↳ DFS O/p = 50 20 10 5 15 45 30 90 85 75 50 100 95 120.
Traversal

BFS ~~Depth~~ 1

BFS Breadth First Search

It is also called as level wise traversal.

↳ ∴ 50, 20, 90, 10, 45, 85, 100, 5, 15, 30, 75, 95, 120

first level 0 then from left level 1 to RBht side.

↳ 1. Preorder (Visit Left & R) ↳ 2. Inorder (LVR) ↳ 3. Postorder (LRV)

1) Traversal preorder

↳ 50, 20, 10, 5, 15, 45, 30, 90, 85, 75, 50, 100, 95, 120

first 50 then go to its left subtree so on till 5. Then go to its parent ~(10) having RBht subtree (15) 15 is having left subtree? No then go to it parents 15 -> 20 -> then 20 having Left subtree NO then go to RBht subtree 45, then 30. And so on traversal done.

2) Inorder Traversal

↳ 5, 10, 15, 20, 45, 30, 50, 50, 75, 85, 90, 95, 100, 125

start from root node without visiting & go to its left subtree until its zero so (5). After (5) does it have R-Sub-Tree? No then visit (10) is 10 having RBht SubTree? Yes (15) as it is empty go to parents then (20) then same way (45), (30) visit again parents ie, (50) so on go ahead.

↳ Inorder o/p always in Ascending Order *

3) Postorder Traversal:

↳ 5, 15, 10, 30, 45, 20, 50, 75, 85, 95, 120, 100, 90, 50.

First visit leftsubtree till its empty ie (5) go to 10 as it has RST ie (15) as it has empty R&L BT now visit (10) First go to parent, but 20 having R-S-T 45 then to go 30 as 30 is empty R/LST then (30), (45), (20) & so on)

↳ Root always comes at last.