DSA

AFTER MIDS

Reg #01

RAYYAN SHABBIR

BITF20M535

STL

★ STL : standard Template Library.

★ It is a library in C++.
★ It is a "template."
★ It is a "generic library" which works on "any data type."
★ In this library there is a lot of header files which contains different DATA STRUCTURES.

★ This library implements in Cpp and alot of "DATA STRUCTURES" already implemented in this.

⇒ **BENEFITS OF STL:**

★ **REDUCE YOUR C/C++ PROGRAMMING TIME:** (Use Data Structures and algorithm from STL directly, instead of implementing everything from scratch.)

* EASIER MAINTAINANCE:
(Bugs in bottom structures are the most difficult to find.)

STL CONTAINERS

➡ SEQUENCE CONTAINERS: (LINEAR)

- list (we can also change size of a list)
- Stack
- Queue
- .
- .
- .

➡ SORTED ASSOCIATEVE CONTAINERS:

- Set (sets only contains unique values; no duplicate values; it also manges their sequence like :: ✓ 1 2 3 4
  ✗ 1 1 2 3)
- multiset
- map
- multi map

➡ STL ALGORITHMS : (MANY GENERIC ALGORITHMS)

- for_each () (It works for each value of list or array.. It does n require size. It simply w for all elements; 0 ... n
- find(), sort(), nth_element()
- count(), count_if(), min(), max()
- copy(), swap(), replace(), remove(
- partition(), merge()
- .

★ If we want to approach a problem by using "stack" OR "queue" we simply include these libraries:

→ If we want to use "STACK" then simply include:

```
#include <stack>
```

↳ ∴ as it is template so, we must define its type in main when we create its object like:

```
stack <type> st;
```

→ If we want to use "QUEUE" then simply include:

```
include <queue>
```

↳ ∴ as it is template, so we must define its type whenever we creates its obj in main:

```
queue <type> q;
```

NOTE: ★ Both "STACK" and "QUEUE" have '10' constructors including copy constructor in STL.

"STACK" FUNCTIONS IN STL

- st.empty() } return TRUE (1) if stack is em else FALSE (0)
- st.pop() } return type void ; nothing ret
- st.push() } return type void ; only push
- st.size() } return type "int" bcz is an integer.
- st.swap() } swapping b/w two stacks.
- st.top() } return value of top return type T (templa

STACK CODE STL

```
#include "iostream"
#include "stack"          // COMPULSORY
using namespace std;

int main() {

    stack st;             // ✗ ERROR ... bcz type is undefined. (As, it is template so, we must have to write its type).

    stack<int> st;        // ✓ VALID
                          //   ↳ By default constructor.

    cout << st.size();
```

OUTPUT: 0 → bcz there is no element in stack thus, its size is '0'.

```
    // st.top();   } PROGRAM CRASH.
    // st.pop();   } (reason next page)
```

OR

NOTE:

* When stack is "empty" and we are trying to 'GET TOP' or 'POP' anything

```
st.size();   ⇒ '0'
st.top();
st.pop();
```

} ⇒ then our program will crash. **ERROR.**

```
    st.push(5);
    cout << st.size();
```

OUTPUT:- '1' → bcz '1' element is pushed into stack.

```
    st.push(2);
    st.push(3);
    cout << st.size();
```

OUTPUT:- '3' → As, total '3' elements in stack

```
    st.pop();
    cout << st.size();
```

OUTPUT:- '2' → bcz now '2' elements as '1' is popped

→ TO GET TOP:

```
cout << st.top();
```
→ Top only read not popped
→ return type is int.

OUTPUT: '2'

→ TO DISPLAY WHOLE DATA OF STACK AND ALSO POP THAT HAS BEEN READ OUT.

→ return type is "bool".

```
while (! st.empty())
{
    cout << st.top();
    st.pop();
}
```

OUTPUT:- '2' ← TOP
'5'

QUEUE FUNCTIONS IN STL

* In STL, queue is known as "DEQ" → "Double Ended Queue".

NOTE:

- Here in queue, for "ENQUEUE" name is "P
  ↳ AT REAR.
  q.push(); → "ENQUEUE".
- for "DEQUEUE" name is "POP".
  ↳ AT FRONT.
  q.pop(); → "DEQUEUE".
- Here, For FRONT → "Front".
  q.front();
- For REAR → "back"
  q.back();
- q.emplace();
- q.container();

} • Read it by yo
if needed.

QUEUE CODE STL

```
#include "iostream"
#include "queue"
using namespace std;

int main() {

    queue<int> q;

    cout << q.size();
```

OUTPUT: '0' → bcz there is no element in queue i.e size=0.

```
// q.pop();
// q.front();
// q.back();
```

} PROGRAM CRASH... ERROR.

NOTE:

* If queue is empty and we by to pop, "POP", get "FRONT", or get "BACK" then our program will crash.

```
q.push(1);
q.push(2);
q.push(3);

cout << q.front();
cout << q.back();
```

} read only front and back (rear), not pop or anything else.

OUTPUT: '1' → front
'3' → back

```
q.pop();
cout << q.front();
cout << q.back();
```

} → pop from "FRONT".

OUTPUT:

'2' → FRONT
'3' → BACK

NOTE:

* As, it(STL) is template (library), so it will work for your own data as well as for other standard data type.

EXPRESSION EVALUATION

ALGEBRIC EXPRESSION :

* Every expression is a legal combination of
  OPERANDS → a, b, c, . . .
  └ OR CONSTANTS → 1, 2, 3, . . .
  and
  OPERATORS → +, −, *, / . . .
* These expressions must be legal like.
  a + b ✓ ; a * + − d / e ✗

* Expressions have forms:
  (i) Infix
  (ii) Prefix
  (iii) Postfix

① INFIX EXPRESSION : { ALGEBRIC FORMS

* Expression in which operands surround the operator
* Operator lies b/w operands.
* We will see position of operator.

Example:-

a + b
a + b * c
a * (b − c) / d

② PREFIX EXPRESSION :

* also known as "POLISH NOTATION" (PN)
* In prefix notation, operator comes before the operands.
  ↳ Operator will be pre / start.

Example:-

+ ab
+ * ab
+ ab * c
* ab + cd

③ POST FIX EXPRESSION:

* also known as "REVERSE POLISH NOTATION" (RPN).
* Operator comes after the operands.
* Operator will be post / last.

Example:-

ab +
abc * +
abc + *
abc * c +
ab + c *

NOTE:

* Infix expressions are easy for paper work only.
But difficult on / for machine.

* So, to make it easy on machine we have two diff / more expressions.
These are easy to evaluate.
(1) POSTFIX.
(2) PREFIX.

DISADVANTAGES OF INFIX:

* Every compiler has some rule of
(1) OPERATOR PRECEDENCE:
• When an operand lies b/w two operator the operand associates with the operator having high priority.

Example: A + B * C
① ②
• The '*' will be evalu 1st followed by '+'

(2) OPERATOR ASSOCIATIVITY:
• When an operand lies b/w two ope having same priority, the operand associate with the operator on left.

Example: A - B + C
① ②
• The '-' will eval 1st followed by '+'

* But these precedence will be mos affected by brackets '( )'.
Bcz braces takes more priority.

* So, here compiler will have to scan whole expression (infix) which take 1st priority and so on.

A - B + C

→ scan till last expressi

* Infix notations are not as simple as they seem specially while evaluating them.
* To evaluate an infix expression we need to consider "OPERATOR PRIORITY" and "ASSOCIATIVE PROPERTY".
* This makes computer evaluation more difficult than is necessary.
  - If we remove both 'PRECEDENCE' and 'ASSOCIATIVITY', then compiler works faster.
* Thus, for this purpose we use "prefix" and "postfix" expressions.
* Both "prefix" and "postfix" have advantage over infix that while evaluating an expression in "prefix" or "postfix" form we need not consider "PRIORITY" and "ASSOCIATIVITY".
* So we can say that either the "prefix" and "postfix" have no "PRECEDENCE" and "ASSOCIATIVITY".

OR

In these all operators have same "PRIORITY" and OR "ASSOCIATIVITY".

* By this performance of our compiler will enhance.

Also, to evaluate these forms of expressions is easier.

CONVERSION

1) INFIX TO POSTFIX:

① MANUALLY / SIMPLE:

* **Basic Algorithm:-** Pick two operands and merge them and then write the operator.

* **Conversion:-** For conversion we must ~~remember~~ remember and follow associativity and precedence.

EXAMPLES:

1) a + b

ab+

2) a * b + c

(② a * b, ① ab* + c)

ab*c+

∴ for conversion we must follow/ remember associativity and precedence.

3) a + b * c

(② b * c, ① a + bc*)

a + bc* (bc* = T)

a + T

aT+

abc*+ (∴ T = bc*)

④ a + b - c

(② a + b, ① ab+ - c)

ab+ - c (ab+ = T)

T - c

Tc-

ab+c- (∴ T = ab+)

∴ Here we check follow associativity "LEFT" to "RIGHT"

* If we do "WRONG WORK"; As both h same priority, So we don't follow associa

a + b - c

(① b - c)

∴ But here a + bc-

compiler will confuse abc-+ → no effect / no c

Bcz As we know; a+b = b+a

Thus, compiler ambiguity comes;

[LIKE DIAMOND PROBLEM]

2) Infix To Prefix:

② Manually / Simply:

* Basic Algorithm:- Pick two operands and merge them and write the operator before them.

* Operator at start / first / pre.
* Operands are at last / post / end.

* While converting infix to prefix manually or by using stack always remembers / follow associativity and precedence.

EXAMPLES:

A + B ⇒ +AB

A * B + C ⇒ +*ABC

(2) A * B + C
(1) *AB + C

③ A + B * C
(2) (1)
A + *BC
+A*BC

④ A + B - C (same sign: operator)
(2) (1)
+AB - C
-+ABC

⑤ A + (B - C) * D
(2) (1) (3)
A + -BC * D
A + *-BCD
+A*-BCD

⑥ (A + (B - C)) * D
(2) (1) (3)
(A + -BC) * D
+A-BC * D
*+A-BCD

RECURSION

* A function calling / defines itself is doing recursion.

* A function is made to solve a "problem."
  A function calling itself means it is solving its own "subProblems."

* **RECURSIVE DEFINITION:** "A recursive definition is one which uses the word or concept being defined in the definition itself."

EXAMPLE: ① List: 1, 2, 3, 4 ....
predecessor ... successor ... so on
.... Recursively ....

② Row major order indexing formula.

FOR RECURSION:
We must have:

1) BASE CASE(S):

* Where "Recursion" stops.
* It is also known as stopping Condition, Base set or simple cases.
* **BASE**, where some simple case(s) of the item being defined are given.

* These are simple cases of those subproblems whose further more division is impossible.
  ↳ These can also be more than '1'

* Base Case (non-recursive) part is necessary otherwise there will be no terminating part and function will become "infinite ~~fine~~ recusive function" → PROGRAM CRASH

• **IN FACTORIAL EXAMPLE:**

fac(3) → fac(2) → fac(1) → Ba...

2) RECURSIVE CASE(S):

* All other callings, then base cases are "Recursive callings."
* Make a call to itself.
* A function/method can call itself; if set up that way, it is called "RECURSIVE METHOD."

APPLICATIONS / EXAMPLES OF RECURSION:

* Exit from a maze.
  ↳ Exit point is a BASE CASE.
* Back Tracking (chess).
* Image processing.
* In Graphs, Trees we use/do recursion because here path is involved.

NOTE:

↳ When we do recursion → The Stack automatically involve

* Bcz when functions calls then all callings pushes into the stack until we reach "base case" and then started popping functions.

→ Each call to the method sets up a new "execution environment", with new parameters and local variables.

• As, recursion is very costly in respect of memory (MEMORY EXPENSIVE) → • So, we not impl it in evey proble → we only apply it when there is must n of it. Otherwise we can also apply "Itera

V. Impt

* As with any method call, when the method completes, control return to the method that invoked i (that called it)

n@>0

① EXAMPLES: TO PRINT NUMBER IN ASCENDING ORDER.

→ 1 , 2, 3 , 4 , 5

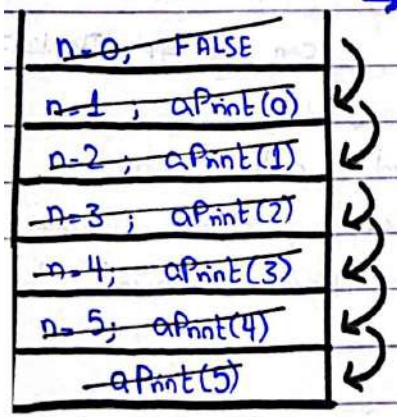
```
void aPrint(int n) {
    if (n > 0)
        aPrint(n-1);
    cout << n;
}
```

→ Not execute until aPrint() (FUNCTION) comes/returns back to its callee.

TRACING RECURSION:

→aPrint(5)
↳ aPrint(4)
↳ aPrint(3)
↳ aPrint(2)
↳ aPrint(1)
↳ aPrint(0)

//0 1 2 3 4 5



STACK

aPrint(5);

② EXAMPLE : 1 TO PRINT NUMBER IN REVERSE ORDER.

→ n , n-1, n-2 , n-3 , . . . , 0
→ 5 , 4 , 3, 2, . . . , 0

```
void rPrint(int n) {
    cout << n; //5 4 3 2 1 0
    if (n > 0)
        rPrint(n-1);
}
```

- If we not apply base case then it is "INFINITE RECURSION"
  - if (n > 0) → Base Case
  - rPrint(n-1); → = Bcz we want to decrease 1 numbr to go reverse.

↳ In visual studio, program crash bcz if not apply base case; It is "Infinite Recursion".

TRACING RECURSIONS:

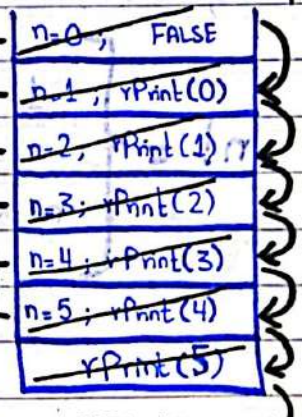
rPrint(0); → 0 , then if check and then stop

rPrint(5); → 5 4 3 2 1 0

main or anything. rPrint(5); → rPrint(4); → rPrint(3); → rPrint(2); → rPrint(1); → rPrint(0)

Ending Part

NOW REVERSE



③ EXAMPLE: FACTORIAL OF A NUMBER 'n'.

For example :- n=3

→ 3! = 3 x (2!) → Calling itself. (subProblem)

→ 2! = 2 x (1!)

→ 1! = 1 x (0!)

↳ must be defined. i.e : 0! = 1.

↳ It is base case.

↳ Where problem ends/stops.

So,

$n! = n \times (n-1) \times (n-2) \times (n-3) \times (n-k) \times 1 \times 0!$ → 1 defined.

$$n! = n \times (n-1)!$$

↳ A problem is defining itself.

→ When a problem defines itself it is a recursion.

So,

RECURRENCE RELATION

$$n! = \begin{cases} 1 & ; \text{ if } (n=0) \\ n \times (n-1)! & ; \text{ if } (n>0) \end{cases}$$

n=5

```
int factorial (int n) {

    if (n == 0)
        return 1;

    return n * factorial (n-1);
```

return ↳

* We apply return here also because we want whatever result comes from recursive function calling fact(n-1); It must be return to its caller function and save for next caller ... so on.

* We always apply return here (In recursive calling) also whenever return type of function is "int" (other than void).

→ So, that value must be return to its caller.

TRACING:

factorial(3);

n=3 factorial(3); → 2*3 → 6

n=2 factorial(2); → 2*1 → 2

n=1 factorial(1); → 1*1 → 1

n=0 factorial(0); → 1

factorial(3) = 6

④ **EXAMPLE:** RETURN SUM OF FIRST 'N' NUMBERS.

→ sum(3) = 3 + 2 + 1

→ sum(5) = 5 + 4 + 3 + 2 + 1

→ sum(n) = n + sum(n-1) + sum(n-2) + sum(n-3) + ··· + 0/1

BASE CASE

↓

0 or 1.

$$sum(n) = n + (n-1)$$

The function is defined in terms of itself.

RECURRENCE RELATION:

$$sum(n) = \begin{cases} 1 ; & \text{if } (n = 1) \\ n + sum(n-1) ; & \text{if } (n > 1) \end{cases}$$

```
int rSum ( int n) {
    if (n <= 0)   OR   if (n == 1)
```

Any of them can be used as "Base case:"

return 0; return 1;

return sum(n-1) + n; → not executes until all function calls finish.

→ not run until function not return back to its caller.

```
}
```

TRACING:

15

sum(5) n=5 → 10 + 5

sum(4) n=4 → 6 + 4

sum(3) n=3 → 3 + 3

sum(2) n=2 → 1 + 2

sum(1) n=1 → 0 + 1

sum(0) n=0 → 0

n=0	Return 0;
n=1	① sum(0) + 1
n=2	① sum(1) + 2
n=3	③ sum(2) + 3
n=4	⑥ sum(3) + 4
n=5	⑩ sum(4) + 5

0 → 1 → 3 → 6 → 10

15 ANS

⑤ EXAMPLE: TO PRINT ARRAY ASSCENDING.

→ Last Index.

```
void print(int A[], int n)
{
    if( n > 0)
        print(A, n-1);

    cout << A[n];
}
```

→ In recursive type functions, always pass here last Index, not the size of array. So, that it will be easy for us to deal with indices of array to get any element.

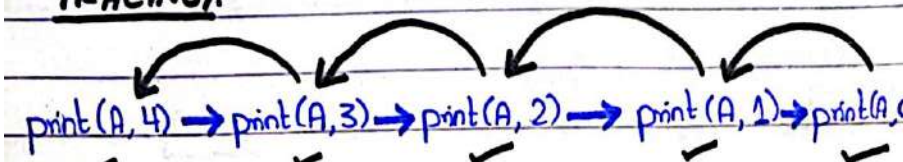
MAIN: print (A, 4);

→ Last Index; Overall size = 5.

★ If user only pass size from main, then we have to make a function "void Oprint (Array[], size);" This function then calls to our Recursive function as: ~~print (Last AE)~~ print (A, size-1);

★ Abstraction from user.

TRACING:



⑥ EXAMPLE: SUM OF ARRAY RECURSIVELY.

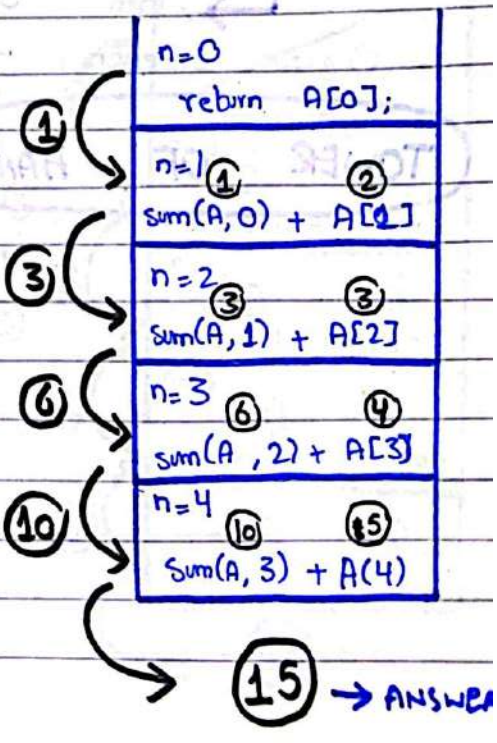
```
int arraySum(int A[], int n) {
    if (n == 0)  return A[n];
        OR
    if (n <= 0)  return A[n];

    return (arraySum(A, n-1)) + A[n];
}
```

→ Last Index. i.e (size-1)

TRACING:

A[] = { 1, 2, 3, 4, 5 } (0 1 2 3 4)

arraySum(A, 4);



NOTE:

⇒ WHY WE DO/USE RECURSION ?

* As array is a bigger problem. So, when we do recursion its size decreases by '1' until Last element.

→ "DIVIDE AND CONQUER RULE":

↳ Divide the bigger problem into smaller until no more division is possible.

↳ Then Solve separately each small problem.

TOWER OF HANOI

① **EXAMPLE:** Sum of squares of 1st 'n' +ve integers

$sum(4) = (4)^2 + (3)^2 + (2)^2 + (1)^2 + (0)^2$

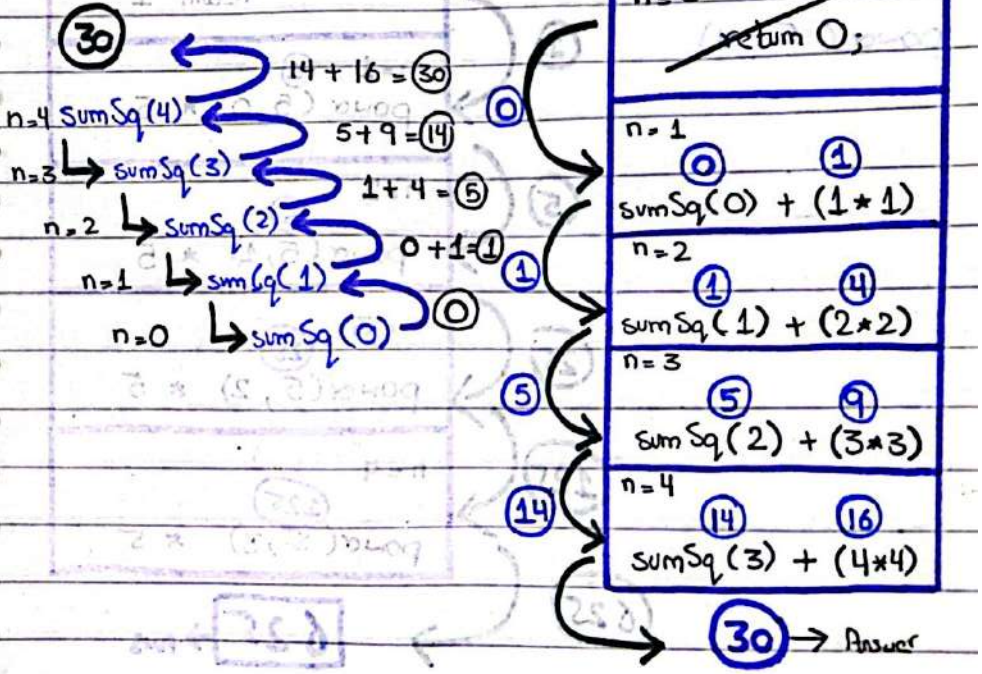
$sum(4) = 16 + 9 + 4 + 1 + 0$

So,

$sum(n) = n^2 + (n-1)^2$

```
int sumSq( int n ) {
    if ( n == 0 )  return 0;

    return sumSq(n-1) + (n*n);
}
```

TRACING: sumSq(4)



RECURSIVE SOLUTION OF LINEAR SEARCH

* Linear Search is a sequential search.
  ↳ In this technique;
  • an ordered or unordered list will be searched one by one from begging or from end.
  * If desired element is found, the search is "SUCCESSFUL".
  * IF desired element is not found in the list, the search is "UNSUCCESSFUL".

CASES:

It has two base cases:

1) When no element found, by searching whole array ⟶ RETURN -1; n < 0
2) When value finds ⟶ RETURN INDEX(n);
3) Recursive function calling to find element.; n>=0.

RECURRENCE RELATION:

$$\begin{cases} n < 0 & ; \text{return } -1; \\ \text{if } a[n] == key & ; \text{return } n; \text{ (index).} \\ n >= 0 & ; rLinearSearch(a, n-1, key); \end{cases}$$

TWO TECHNIQUES OF SEARCHING IN L.S:

- n-1 ⟶ 0 indices
- 0 ⟶ n-1

* Here we will search from "n-1 index to 0":

NOTE: if size = 6;
→ LinearSearch (A, 6, 3); } Exposure to user
↳ Array bandy checks etc handles in this function, not in recursion.

→ rLinearSearch (A, 5, 3);
5 → LAST INDEX.
3 → KEY ELEMENT.

```
int rLinearSearch (int A[], int n, int key) {
    if (n < 0)
        return -1;
    if (A[n] == key)
        return n;
    return rLinearSearch(A, n-1, key);
}
```

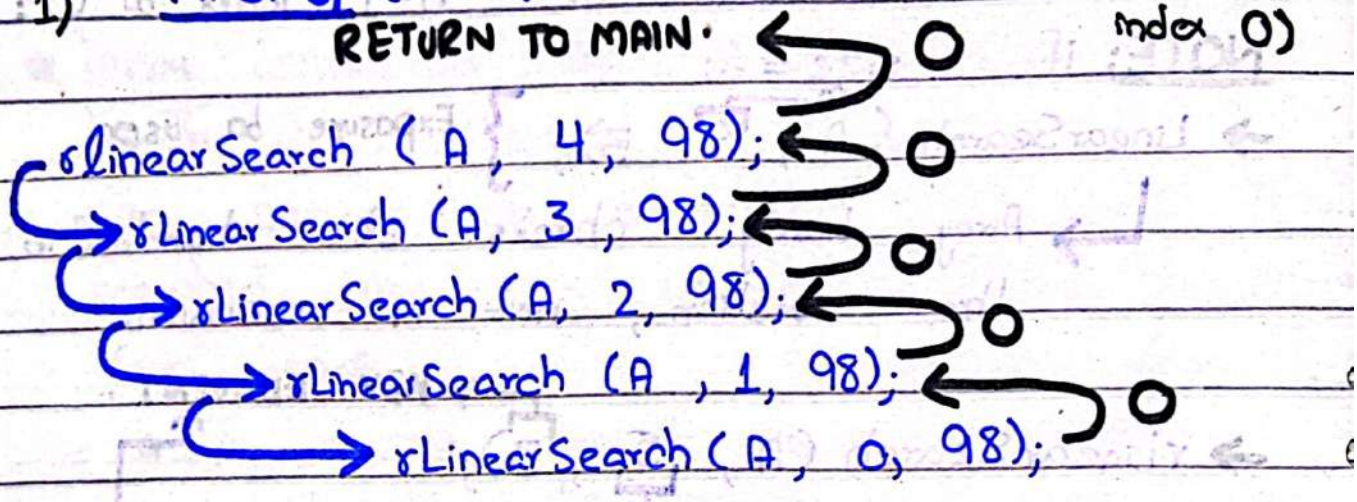
- int: return index
- Sequence Compulsory matters. Otherwise when no element found 'n = -1' if 1st & 2nd 'if' apply 1st then → A[-1] → OUT OF BOUND.
- if (n < 0) return -1; → key element does not exists
- if (A[n] == key) return n; → key element found at 'n'.
- return rLinearSearch(A, n-1, key); → This return is the return which will return to "main". It executes in Last.

TRACING LINEAR SEARCH

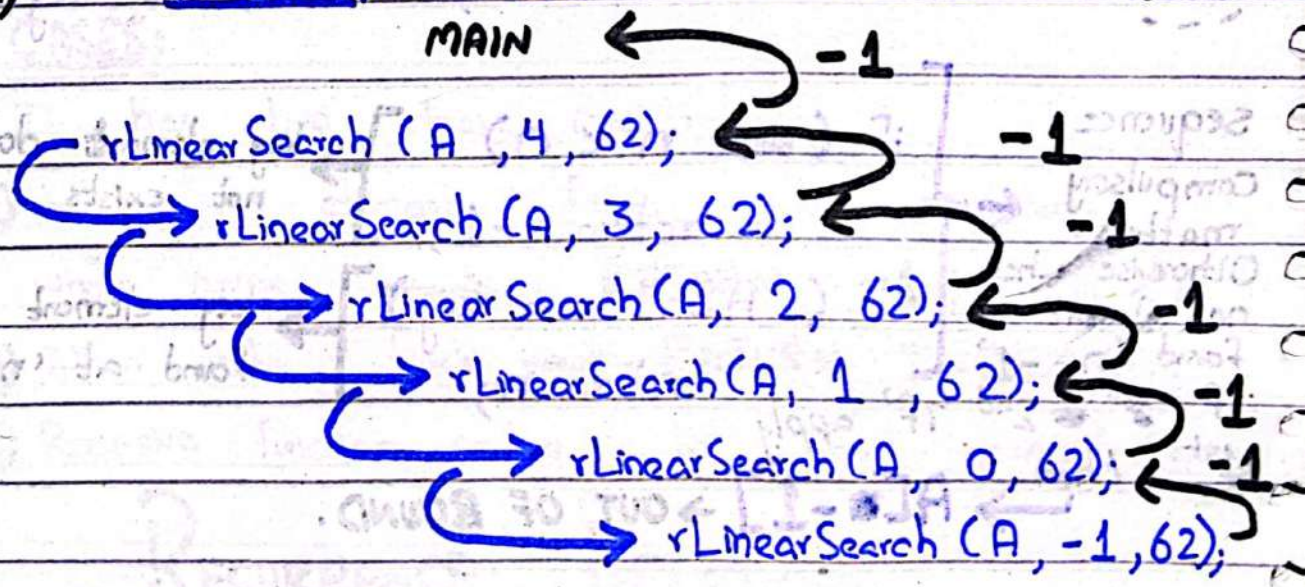
0	1	2	3	4
98	6	7	8	23

A[] = {98, 6, 7, 8, 23};

1) FINDING : 98 (element found at index 0)



2) FINDING 62 (No element found)



TIME COMPLEXITY:

* Time complexity of recursive Linear Search is also $O(n)$.

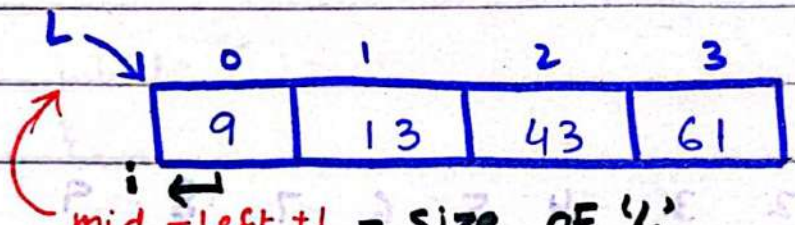
MERGE SORT

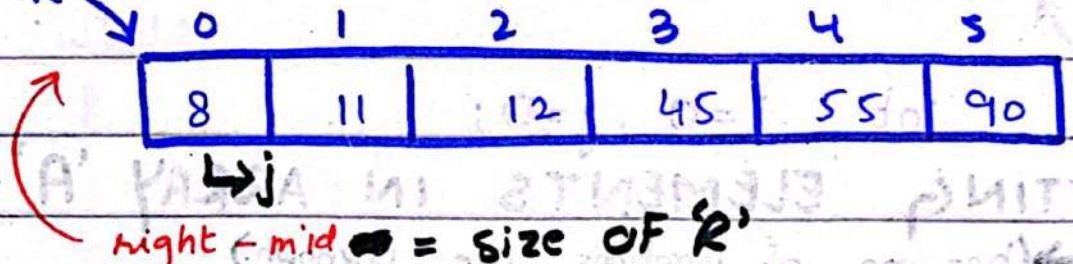
* Merging is a process in which two sorted lists are merged together in a list. (Merge Sort has Recursive nature).
* Merge Sort is based on the proces of Merging.
* Sort by using merge.

EXAMPLE:

MERGE ALGORITHM:

Let's take two sorted lists.





→ To merge these lists, we need extra space. (Extra memory for result)

→ We have to create an array of size, that can occupy merged list.

Code of Merge:

CREATING ARRAY:

*When we don't know size of array we can use this formula to find size of array.

```
int n_L = size of (L) / size of (L[0]);
int n_R = size of (R) / size of (R[0]);
int n_A = n_L + n_R;
```

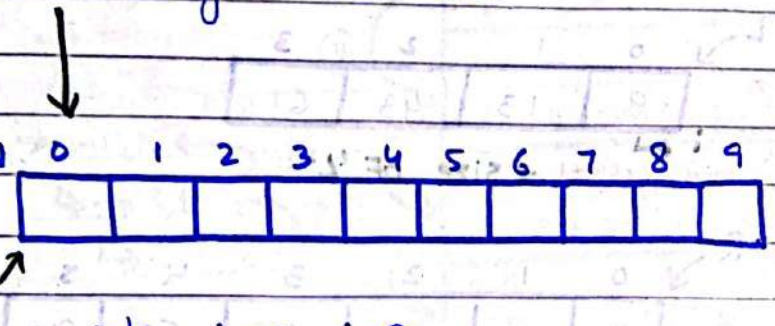
↳ also use here sizeof(int) but that will be only for int array. not for char, string etc.

```
int * A = new int[n_A];
```

Basically, In code:

```
int L_Size = mid - left + 1;   → for including mid element.
int R_Size = right - mid;      → here not including mid element.
```

The logical representation of this new array will be:




```
int i=0, j=0;
```

PUTTING ELEMENTS IN ARRAY 'A':

(bcz no. of iterations are unknown)

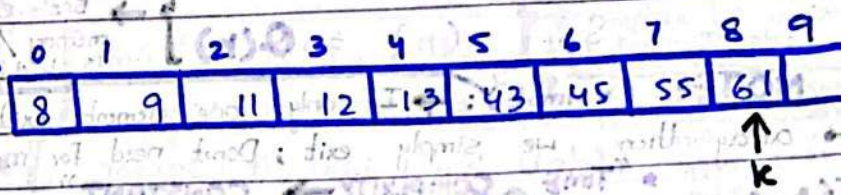
```
while ( i < n_L && j < n_R)
{
    if (L[i] < R[j])
        A[k++] = L[i++];
    else
        A[k++] = R[j++];
}
```

L_Size ← or → R_Size

} equals handles here

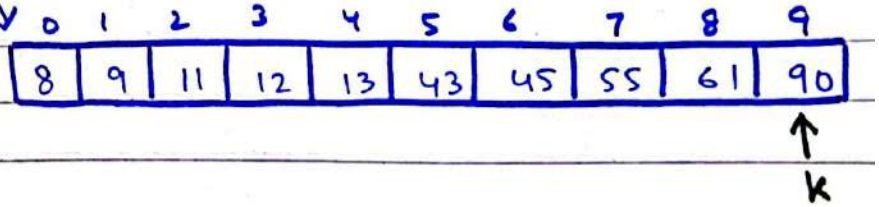
* Through this algorithm, elements will be arranged as:

(∴ Here size of 'L' array finish)



* For putting last element:

```
while (i < n_L)
{
    A[k++] = L[i++];
}
while ( j < n_R)
{
    A[k++] = R[j++];
}
```



* Merging algorithm will have complexities:

- **TIME COMPLEXITY:**

$t_{merge}(n) = \Theta(n)$ } → Bcz whole array traverse.

↳ In best, worst and average case

- **SPACE COMPLEXITY:**

$S_{p_{merge}}(n) = \Theta(n)$ } → Bcz extra memory/space is build/required.

NOTE: (v.impt): → If only one element exists in array then we simply exit; Don't need for moging it.

• "TIME COMPLEXITY → CONSTANT"

"MERGE SORT" ALGORITHM:

* It works on "divide and conquer" rule. (*use "DIVIDE AND CONQUER RULE".)

EXAMPLE:

Let's suppose we have an array:

0	1	2	3	4
5	3	4	1	2

ALGORITHM:

start index ↑ end index ↑

```
void ms(int A[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right-left)/2;
        ms(A, left, mid);
        ms(A, mid+1, high);
        merge(A, left, mid, right);
    }
}
```

bcz it Divides; so takes "$\lg_2 n$" → $\lg_2 n$

n

First Left half is complete by divide Left half. It also includes mid.

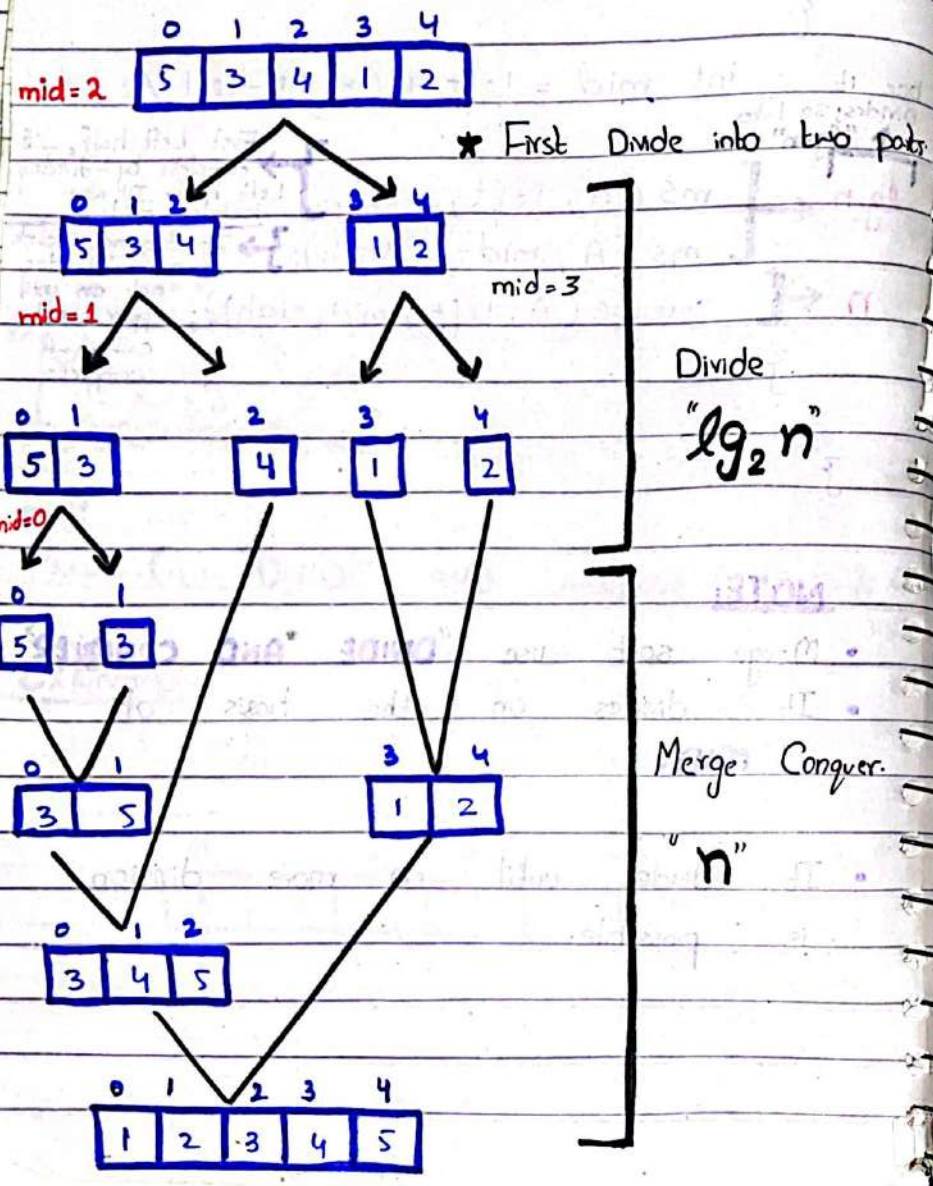
It does not include mid. It do not run until upper Left calling will complete.

NOTE:

- Merge sort use "DIVIDE AND CONQUER"
- It divides on the basis of MID.
- It divides until no more division is possible.

mid = 3 + 1/2

LOGICAL REPRESENTATION OF "MERGE SORT : RUN:"



COMPLEXITIES OF MERGE SORT:

→ **Time complexity:**

$t(n) = \Theta(n \lg n)$ } • 'n' → merging
• 'lgn' → Dividing

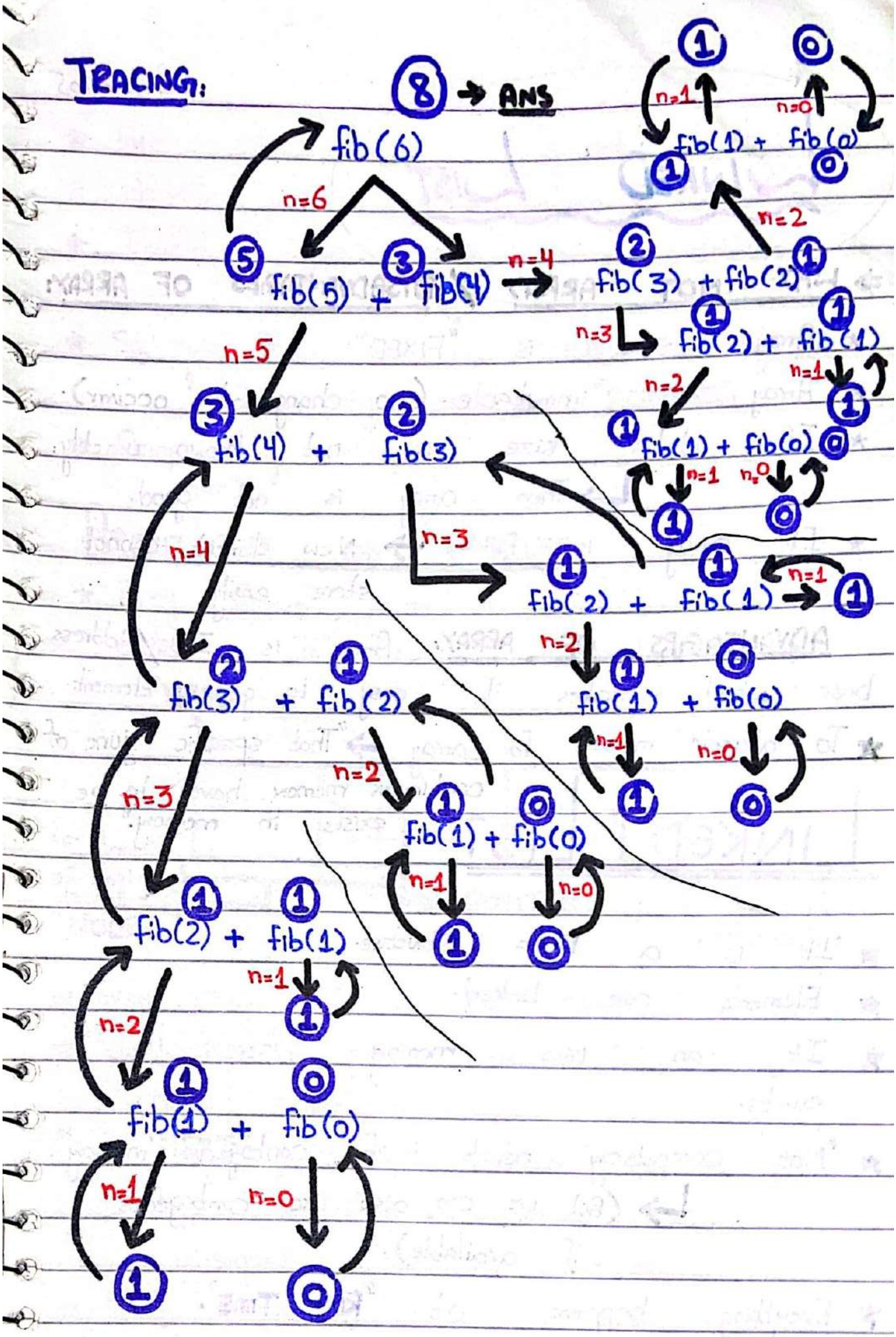
→ **Space complexity:**

$Sp(n) = \Theta(n)$ } As, it use merging algo

↳ • Due to this reason merge sort cannot be used for BILLION of data but can be used for MILLION of data.
• Not used for huge data.

MERGE SORT → Develops in "1951".

↳ **APPLICATIONS** • Sector Designing.
• Block Designing..



Lecture #05

LINKED LIST

⇒ WHY NOT ARRAY / DISADVANTAGES OF ARRAY:

* Array size is "FIXED".
* Array is immutable. (no changement occurr).
* If data size is not known exactly,
  ↳ Then array is not good.
* If array is FULL → New element cannot store easily.

ADVANTAGES OF ARRAY: Array is Index/Address base which makes it easy to get an element.

* To allocate memory for array → "That specific junc of continuous memory have to be exists in memory."

LINKED LIST

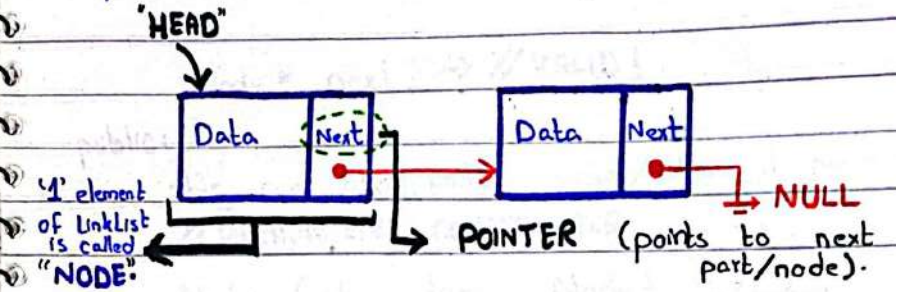
* It is a Data Structure.
* Elements are linked.
* It can uses memory where it exists.
* "Not compulsory needs of contageous memory".
  ↳ (But we can also use contageous if available).
* Evrything happens at "RUN TIME".

* Size of LinkList is not fixed.
* Element comes → memory allocates.
* Element goes → memory deletes.

* So, we have to use "HEAP" for LinkList.
  ↳ That's why we use "POINTERS".
* Comparatively slow.

ARCHITECTURE OF LINK LIST:

* Every element / Node in LinkList have two parts. (i) Data (ii) pointer (next).

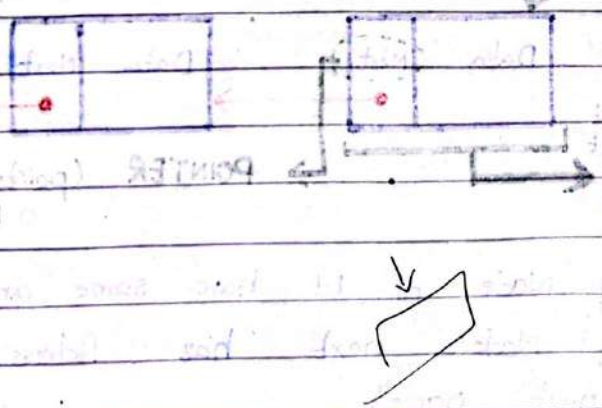


* Every Node of L.L have same architecture.
* "Every Node's next has Adress of next node".
  → "NEXT" is a pointer of a node type which is pointing towards subsequent (successor) node.
* Last Node's next has NULL → shows "END" of a list.

* In LL we must have HEAD (pointer) which stores/holds starting Node OR starting adress of node.
* "Head" type must be same as "NODE" (Overall element type)

"HEAD" is a pointer of node type, which holds starting Node (ADRESS).

→ "Datatype of pointers would be same as of Node".



CODE OF LINK LIST

class Node { } → This class does not do any operation/working, It only holds data.

(class can also be used struct.)

→ programitically name will be Student, Teacher etc.

```
int data;
Node next;
```

→ // ERROR! A class cannot create its own object as data member inside itself. Here copy constructor call or infinitely recursion takes place.

```
Node* next;
```

→ // VALID!

```
public:
    // We can also write getter/setter if needed
    // PARAMETRIZE CONSTRUCTOR
    Node(int data, Node* next)
    {
        this->data = data;
        this->next = next;
    }

    friend class LinkedList;
};
```

→ Bcz LinkedList class has to do working → handles all operations of "Node" class

```
class LinkedList {
```
→ Working class
→ Not practical name. Right name will be student list, processor list etc.

We cannot make it constant.

```
    Node * head;
```
// Bcz whenever List creates by object; We needs "HEAD".

```
public:
    // DEFAULT CONSTRUCTOR (Here necessary)
    LinkedList () {
        head = NULL;
    }
    // TO CHECK LIST IS EMPTY?
    bool isEmpty () {
        return head == NULL;
    }
```
→ Return TRUE if list is empty.

```
        OR
        return !head;
    }
```
// IS FULL function does not make any sense in LL.

TO INSERT AN ELEMENT IN LIST:

* Insertion in a list can be:
  - ↳ At start.
  - ↳ At End.
  - ↳ After Node.
  - ↳ Before Node.
  - ↳ Insert after largest element etc.

// INSERT NODE AT END IN LL
↳ No need to maintain CurSize or MaxSize etc

```
void insertAtEnd (int data) {
    Node * newNode = new Node(data, NULL);
```
Whenever we have to insert an element we requires a newNode which we will create on "HEAP". At this stage this "Node" is not part of our LinkList.

Bcz we are going to insert at end.

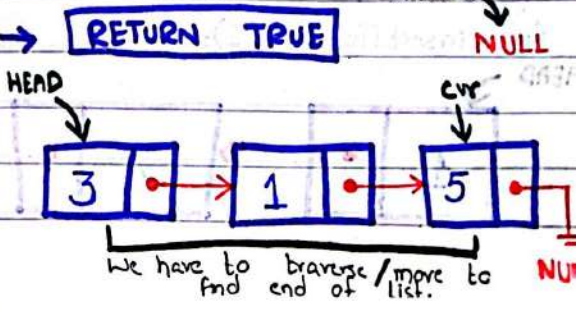
```
    if (head == NULL) {
        head = newNode;
    }
```
Boundry Case, Special Case

```
    else {
        Node * cur = head;
        while (cur -> next != NULL) {
            cur = cur -> next;
        }
        cur -> next = newNode;
    }
```
Bcz we now wants to leave "HEAD"

LINEAR SEARCH

MAIN

```
LinkedList list;
list.isEmpty();
list.insertAtEnd(3);
list.insertAtEnd(1);
list.insertAtEnd(5);
```
→ RETURN TRUE



// INSERT AT START IN LL:

```
void insertAtStart (int data) {
```

It takes Constant time ① $\Theta(1)$. either the list is sorted or unsorted.

```
    Node* temp = new Node(data, NULL);
    temp -> next = head;
    head = temp;
```

OR

simple ②

```
    head = new Node(data, head);
```

It works 1st.

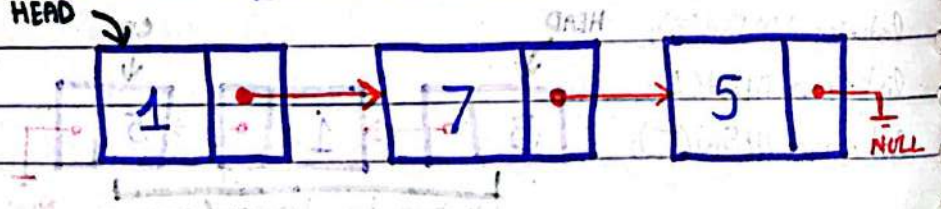
It works late (after creating node) bcz '=' assignment has less precedence.

```
}
```

MAIN

At start: HEAD → NULL

```
list.insertAtstart(5);
list.insertAtstart(7);
list.insertAtstart(1);
```



// INSERT B/W SOMEWHERE:

↳ i.e After that Particular item.

We 1st need to find this Node.

↳ So, it is done by "LINEAR SEARCH"

↳ Thus, it takes O(n)

TRAVERSE IN To ~~UPDATE~~ DATA ~~OF~~ NODE:

* 1st we need to traverse, to find the element which we want to update (i.e LINEAR SEARCH)
  - return ADRESS → type NODE*
  - return TRUE/FALSE → type BOOL

```
Node * Search (int key) {
    Node * cur = head;

    While (cur != NULL) {
        if (cur -> data == key)
            return cur;
        cur = cur -> next;
    }
    return NULL; // OR return cur;
}
```

return the ADRESS of the element if finded.

Bcz cur at this stage is also NULL

WORST CASE: Item not exists if reach NULL.

* TO UPDATE DATA IN LL

TO DISPLAY DATA OF LIST

```
// DISPLAY DATA OF LIST:
void displayData () {
    Node * cur = head;

    if (cur == NULL)
        cout << "Linked List is empty";
    else {
        while (cur != NULL) {
            cout << cur -> data;
            cur = cur -> next;
        }
        // OR
        do {
            cout << cur -> data;
            cur = cur -> cur -> next;
        } while (cur != NULL);
    }
}
```

cur -> next != NULL → FALSE

As, it is not displaying la node's dat

* We can also use do while bcz in else we have gurantee atleast '1' element exists.

Bcz atleast '1' data is available.

DELETING A NODE IN LINKEDLIST:

* To delete a node in LinkList.
  - ① Delete 1st Node.
  - ② Delete Last Node.
  - ③ Delete Particular Node.
    - Depend on Search.

// DELETE LAST NODE.

No need to pass anything.

```
void deleteLastNode () {
    if (head == NULL) cout << "Empty List";
    else if (head -> next == NULL) {
        delete head;
        head = NULL;
    } else {
        Node* cur = head;
        Node* prv = NULL;
        while (cur -> next != NULL) {
            prv = cur;
            cur = cur -> next;
        }
        prv -> next = NULL;
        delete cur;
    }
}
```

Handles Boundry Case Independently.

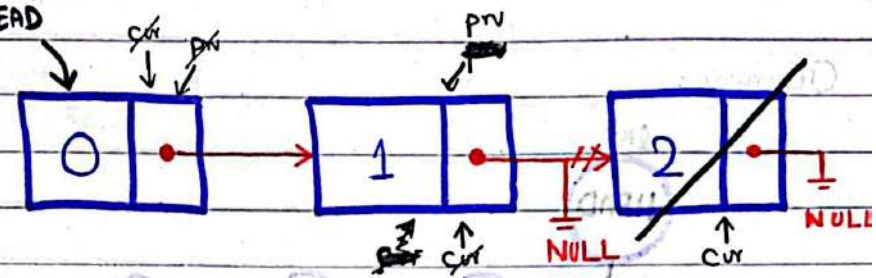
NODE > 1

- Turtle Rabbit Problem.

→ two "POINTERS"

MAIN

```
list . deleteLastNode ();
```



// DELETE AT START (FIRST NODE):

```
void deleteAtStart () {
    if (head == NULL) cout << "Empty List";
    else {
        Node * cur = head;
        head = head -> next;
        delete cur;
    }
}
```

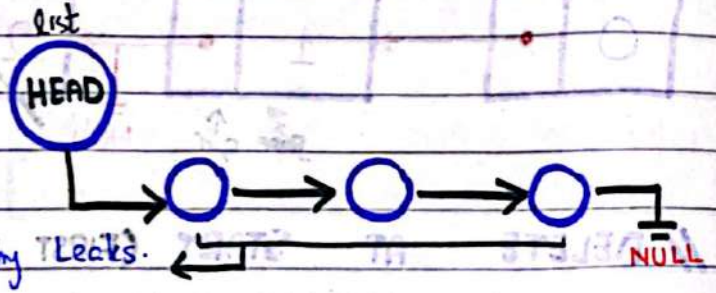
It takes constant time

$\Theta(1)$

★ DESTRUCTOR OF LINK LIST:
(v. IMPT)

★ As, we creates all nodes on heap. So, when List Finishes or scope ends then we must have to delete all **NODES + HEAD** by ourselves in **DESTRUCTOR**.

Otherwise:



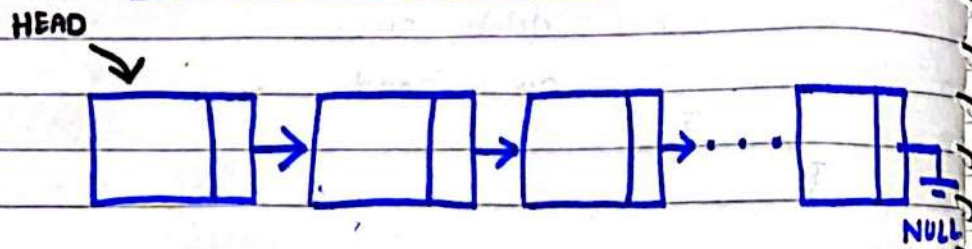
: Memory Leaks.

★ Destructor is necessary whenever we create LinkedList.

```
// DESTRUCTOR
~LinkedList () {
    Node* cur = head;
    while ( cur != NULL) {
        head = head -> next;
        delete cur;
        cur = head;
    }
}
```

TYPES / VARIATIONS OF LINKED LIST

1) SINGLY LINKED LIST:



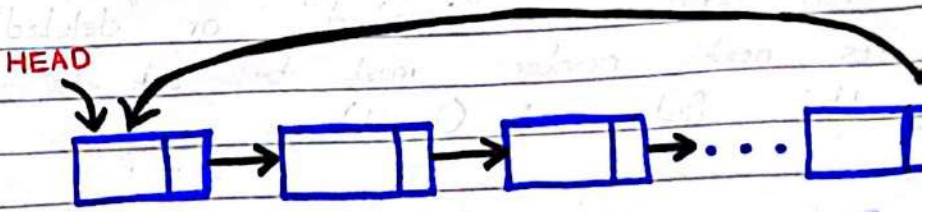
→ It is forward only.

→ It is used when we only / simply want to go forward to traverse one by one each node.

* In Singly Linked List ; "No previous pointer exists."

```
class Node {
    int data;          // HOLDS DATA
    Node* next;        // HOLDS REFERENCE OF NEXT NODE
};
```

2) CIRCULAR LINK LIST:



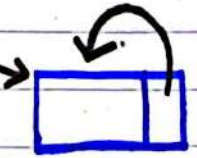
↳ Last node points to head.

↳ Last node is not pointing to NULL.

↳ The "next pointer" of the "Last node" is Linked to the "first node (head

```
class Node {
    int data;          // holds data
    Node* next;        // holds reference of nex
                       nod
};
```

* FIRST NODE IN CIRCULAR L.L: (ONLY NODE)



(self-referenci

```
head->next = head;
```

* FOR EVERY NEW NODE:

```
newNode = new Node (data, head);
```