==========
 # DS DAY-06
+ Limitations of SCCL:
- addition and deletion operations are not effiecient
- we cannot access prev node of any node from it
- we traverse SCLL only in a forward direction.

- and hence to overcome limitations of singly linked list, doubly linked list has beed designed.

+ "Doubly Linked List": it is a collection/list of logically related similar type elements in which:
- head always contains an addr of first node, if list is not empty
- each node contains actual data and an addr of its next node as well an addr of its prev node.

iii. 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 three parts:
1. data part: which contains actual data of any primitive/non primitive
2. pointer part(next): which contains an addr of its next node
3. pointer part(prev): which contains an addr of its previous node
- prev part of first node and next part of last node contains NULL.

+ Limitations of DLLL:
- In DLLL, add_last() & delete_last() functions takes O(n) time, and hence DCLL has been designed.

vi. 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 three parts:
1. data part: which contains actual data of any primitive/non-primitive
2. pointer part(next): which contains an addr of its next node
3. pointer part(prev): which contains an addr of its previous node
- prev part of first node contains of addr of last node, and next part of last node contains an addr of the first node.

- DCLL: add_last(), add_first(), delete_last() & delete_last() operations takes O(1) time.

+ "Difference between array and linked list":
- an array is a "static" data structure, whereas linked list is a "dynamic" data structure.
- addition & deletion operations are not convenient as well as efficient in array as it takes O(n) time, whereas in a linked list addition & deletion operations are efficient as it takes O(1) time, and these operations are convenient as welll.

- array ele's gets stored into the stack section of the main memory, whereas linked list ele's gets stored into the heap section.
- to access array ele's is efficient than to access linked list ele's, as to access array ele's "random access" method is used whereas to access linked list elements "sequential access" method is used.
- to store n no. of ele's in an array takes less space than to store n no. of ele's in a linked list, as in a linked list there is need to maintains link between ele's, whereas in an array link between ele's is maintained by the compiler.

```
arr[0] = *(arr+0)
arr[i] = *(arr+i)
```

- searching operation is an efficient on an array than linked list, as we can apply binary search only on array

+ applications of linked list:
- linked list is used vastly in a system programming
e.g. kernel linked list, kernel data structures
- linked list also used to implement OS algo's
- linked list can be used in any application program in which collection/list of is dynamic.

+ "Stack": it is a collection/list of logically related similar type of elements in which element's can be added as well as deleted only from one end referred as "top" end.
- in this list element which was added last can only be deleted first, so this list works in "last in first out" manner and hence this list is also called as "LIFO" list/"FILO": (first in last out) list.

- we can perform three basic operations on stack in O(1) time:
1. Push : to insert/add an ele into the stack from top end
2. Pop : to delete/remove an ele from the stack which is at top end
3. Peek : to get the value of topmost ele (without Push & Pop)

- by concept stack is dynamic in nature
- we can implement stack by two ways:
1. static stack (by using an array)
2. dynamic stack (by using linked list)

1. static stack (by using an array):

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

```
arr: int []
top: int
```

```
stack_full     : top == SIZE-1
stack_empty    : top == -1
```

1. Push: to insert/add an ele into the stack from top end
step1: check stack is not full
step2: increment the value of top by 1
step3: insert an ele into the stack at top position

2. Pop: to delete/remove an ele from the stack which is at top end
step1: check stack is not empty
step2: decrement the value of top by 1 (i.e. we are deleting an ele from the stack).

3. Peek: to get the value of topmost ele (without Push & Pop)
step1: check stack is not empty
step2: return the value of topmost ele (without incrementing/decrementing top).

2. dynamic stack (by using linked list):

```
push : add_last()
pop  : delete_last()

OR

push : add_first()
pop  : delete_first()
```

+ "Stack Applications":
- to control flow of an execution of programs an OS maintains stack data structure.
- in recursion as well internally an OS maintains a stack data structure
- stack is used to implement undo & redo functionalities in an OS.
- stack is used to implement advanced data structure algorithms like "dfs: depth first search" traversal in a tree & graph.
- stack is used to implement following algo's:
  - to convert given infix expression into its equivalent postfix
  - to convert given infix expression into its equivalent prefix
  - to convert given prefix expression into its equivalent postfix
  - to evaluate postfix expression
  etc...

+ "expression": it is a combination of operands & operators
- there are three ways by which we can mention expressions:
  1. infix : a+b
  2. prefix : +ab
  3. postfix: ab+

infix expression   : a*b/c*d+e*f-g*h
postfix expression : ab*c/d*ef*+gh*-
infix expression   : -+*/*abcd*ef*gh

infix to postfix:
infix : a*b/c*d+e*f-g*h --> string

postfix : ab*c/d*ef*+gh*- --> initially empty string

cur ele =

stack:

- infix expression, empty postfix expression and stack which is initially empty

step1: start scanning an infix expression from left to right

step2:

```
if( cur ele is an operand )
    append it into the postfix expression
else//if cur ele is an operator
{
    while( !stack is not empty && priority(topmost ele) >=
    priority(cur ele))
    {
        pop ele from the stack and append it into the
        postfix expression
    }

    push cur ele into the stack
}
```

step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's from the stack one by one an append them into the postfix expression.

Homework: Saturday & Sunday:
- implement conversion of infix expression into its eq postfix
- implement add_last, add_first, add_at_position, delete_last, delete_first
& delete_at_position functionalities for SCLL, DLLL & DCLL.
Infix : a * b / c * d + e * f - g * h

Postfix :
=> a * b / c * d + e * f - g * h

=> ab* / c * d + e * f - g * h

=> ab*c/ * d + e * f - g * h

=> ab*c/d* + e * f - g * h

=> ab*c/d* + ef* - g * h

=> ab*c/d* + ef* - gh*

=> ab*c/d*ef*+ - gh*

=> ab*c/d*ef*+gh*-

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

Prefix:
=> 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

=> */*abcd + *ef - g * h

=> */*abcd + *ef - *gh

=> +*/*abcd*ef - *gh

=> -+*/*abcd*ef*gh