What are data structures?

Data structure is a storage that is used to store and organize data. It is a way of arranging data so that it can be accessed and updated efficiently. Data structures is the collection of data types arranged in specific order.

- Linear data structures - In this, elements are arranged in a sequence, one after the other.
- Non linear data structures - In this, elements are arranged in a heirarchial manner, where one element connects to more than one elements

ARRAYS

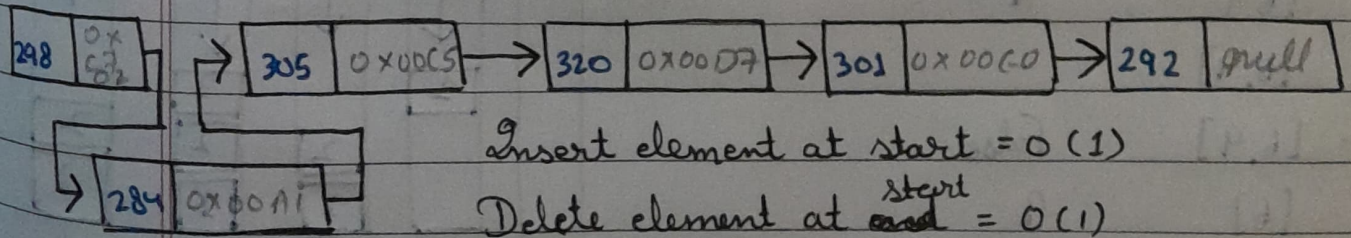
(I) What element at index 'n' LBP = O(1)
(II) What element at index '0'. [TOP] LBV = O(1)
(III) Print all elements Traverse = O(n)
(IV) Insert new ~~price~~ element Insertion = O(n)
(V) Delete element at index '0' [TOP] deletion = O(n)

298	0x80500
305	0x00504
320	0x00508
301	0x0050A
292	0x0050F

→ python is dynamic

```
stock_prices = [2, 3, 5, 6]
stock_names = ["APPLE", "IBM", "TATA"]
stock_data = [{"ticker": "APPLE", "PRICE": 302},
              {"ticker": "IBM", "PRICE": 902}
              {"ticker": "TATA", "PRICE": 278}]
```

arrays can store numbers, text or complex objects



Insert element at start = O(1)
Delete element at ~~end~~ start = O(1)
Insert / Delete at end = O(n)

ABSTRACTION

Notion of abstraction is to distill a complicated system down to its most fundamental parts.

Abstract data type is a model that specifies the type of data stored the operations supported on them and types of parameters of the operation.

STACK (Last in First out)

Insertion (push) `S.push()` has overflow condition O(1)/O(n)

Deletion (pop) `S.pop()` has overflow condition O(1)

S.first() returns index -1 `PEEK` (doesn't remove) O(1)

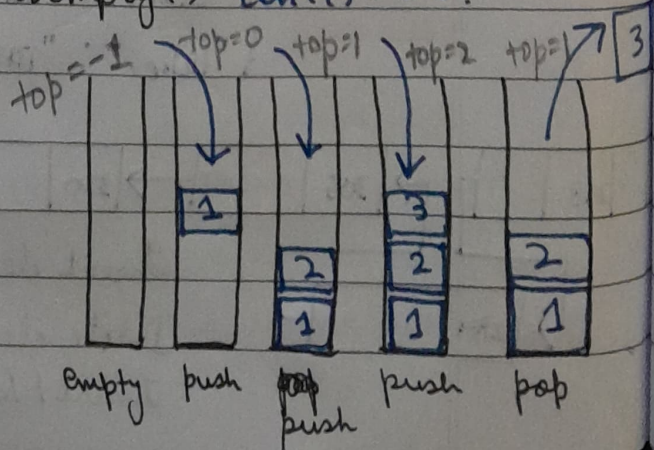
S.len() returns size O(n)

S.isempty checks if stack empty or not. O(1)

S.push(6)	[6]	-
S.push(7)	[6,7]	-
S.push(8)	[6,7,8]	-
S.isempty()	[6,7,8]	False
S.first()	[6,7,8]	8
S.pop()	[6,7]	8
len(S)	[6,7]	2
S.pop()	[6]	7
S.push(9)	[6,9]	-
S.isempty()	[6,9]	False
S.pop()	[6]	9

Implementation using arrays / list

S.push(e)	L.append(a)
S.pop()	L.pop()
len(S)	len(L)
S.first()	L[-1]
S.isempty()	len(L) == 0.



classmate
Date
Page

Working of Stack

1) A pointer, top is used to keep track of the top element, it is assigned the value, [-1]
2) After every push(), the value of top changes. We increase value
3) On popping, we decrease value of top to change it. (explicit declaration)
4) Before pushing, check if already full (OVERFLOW)
5) Before popping, check if empty (UNDERFLOW).

Application of Stack

1) TO REVERSE A WORD put letters in stack, then pop.
2) BROWSER HISTORY

TIME AND SPACE COMPLEXITY O(n)

Asymptotic notations (asymptotes, can be plotted)

Big O worst possible time it can take.
Big Ω best possible time / best case.
Big Θ average case.

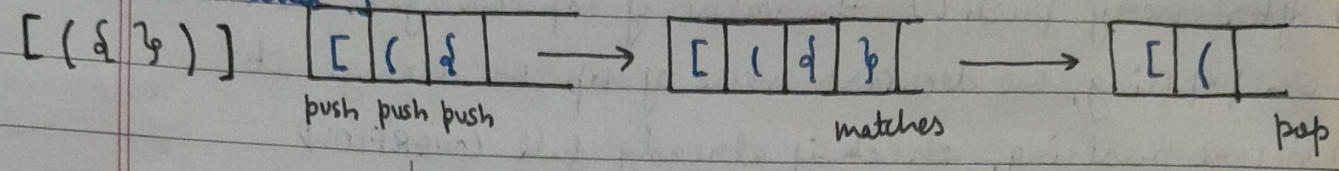
let n = size of inpu

→ space complexity

O(1): Constant type. If elements increase, then ~~space~~ time complexity doesn't increase
O(n): If space complexity (no. of elements) increase, then time complexity increases linearly.
O(n²): Quadratic space time relation.
O(log n): Logarithmic space-time relation.

MATCHING DELIMITERS

Arithmetic expressions that may contain various pairs of grouping symbols; parenthesis : (and). braces : { and }. brackets [and]



~~Logic for opening~~

S.push ([ )	[
S.push (( )	[(
S.push ({)	[({
S.push (})	[({ }
matches	-
S.pop ({)	[(
S.push ( ))	[( )
matches	-
S.pop (( )	[
S.push ( ])	[ ]
matches	-
S.pop ([ )	

```
def is_matched(expr):
    lefty = '({['
    righty = ')}]'
    S = ArrayStack()
    for c in lefty:
        S.push(c)
    elif c in righty:
        if S.is_empty():
            return false
        else if righty.index(c) != lefty.index(S.pop()):
            return false
    return S.isempty()
print('is_matched')
```

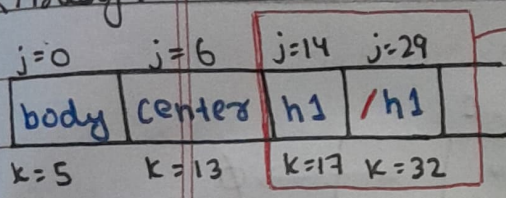
Finding pairs in HTML tags.

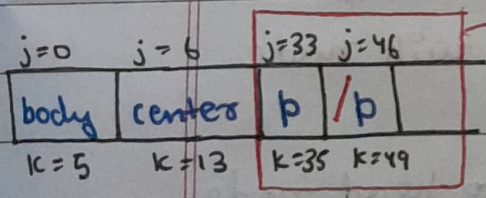
Find first '<' tag. Then find '>' character. If found, then extract the tag.

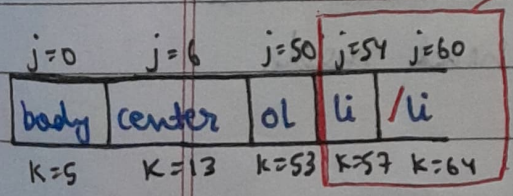
If tag doesn't start with '/', it is an opening tag. If it starts with '/', it is a closing tag.

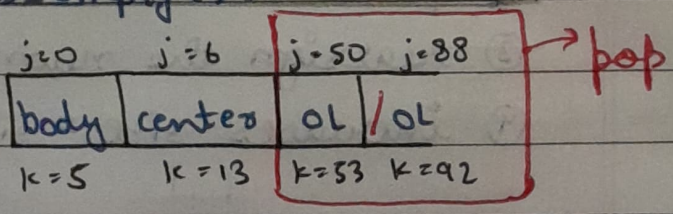
```
<body>
<center>
<h1> The little Boat </h1>
<p> Hello World </p>
<ol> <li> 01 </li>
     <li> 02 </li>
     <li> 03 </li>
</ol> </center>
</body>
```

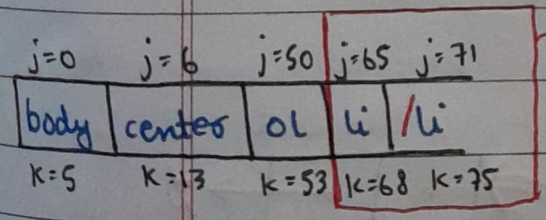
```
S = Array Stack []
j = raw.find('<')
while j != 1:
    k = raw.find('>', j+1)
    if k == -1:
        return False
    tag not tag.starts with ('/'):
        S.push(tag)
    else:
        if S.is_empty():
            return False
        if tag[1:] != S.pop():
            return False
    j = raw.find('<', k+1)
return S.is_empty()
```

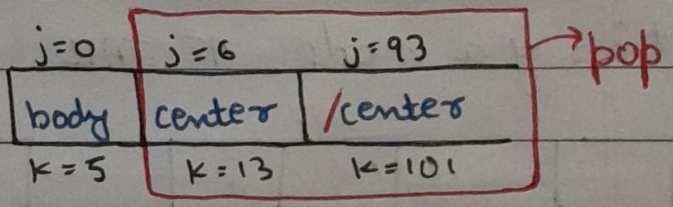


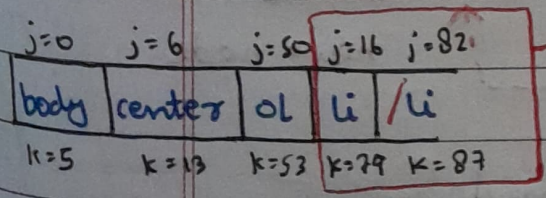


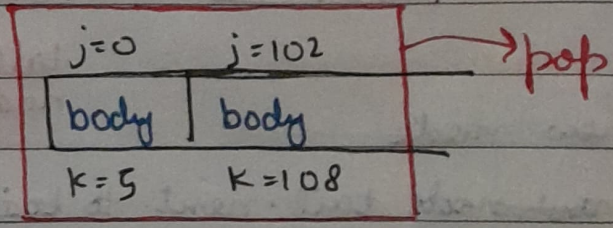






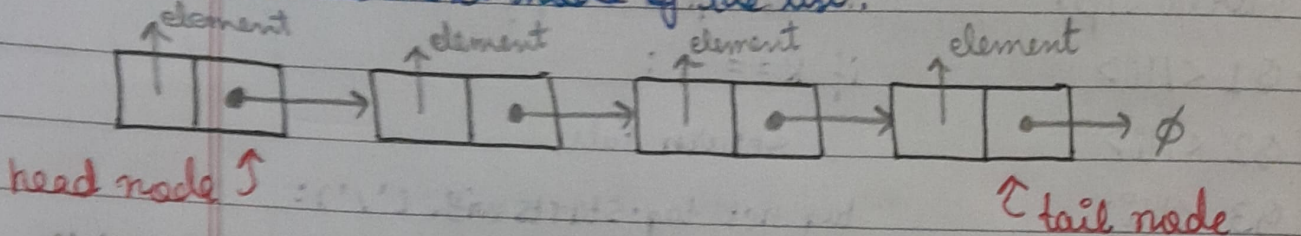






LINKED LIST

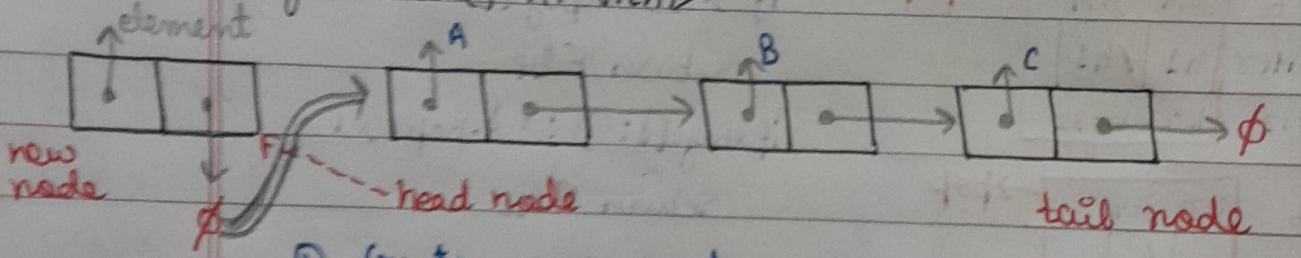
Singly linked list is a collection of nodes that forms a linear sequence. Each node stores reference to its element object, as well as to the next node of the list.



this is explicit declaration

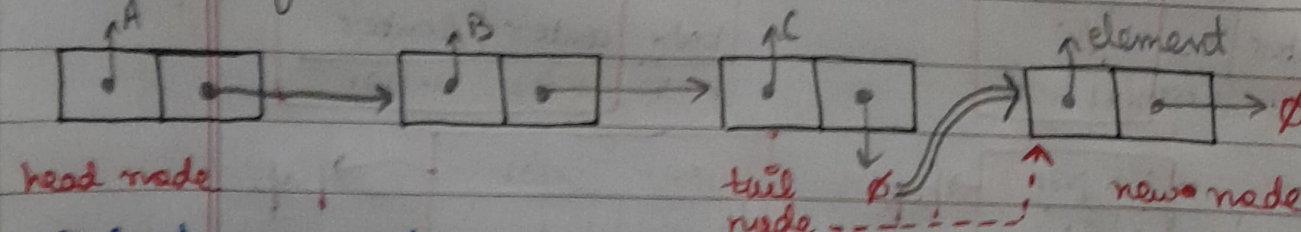
Insertion ——

Insertion of a node, AT HEAD



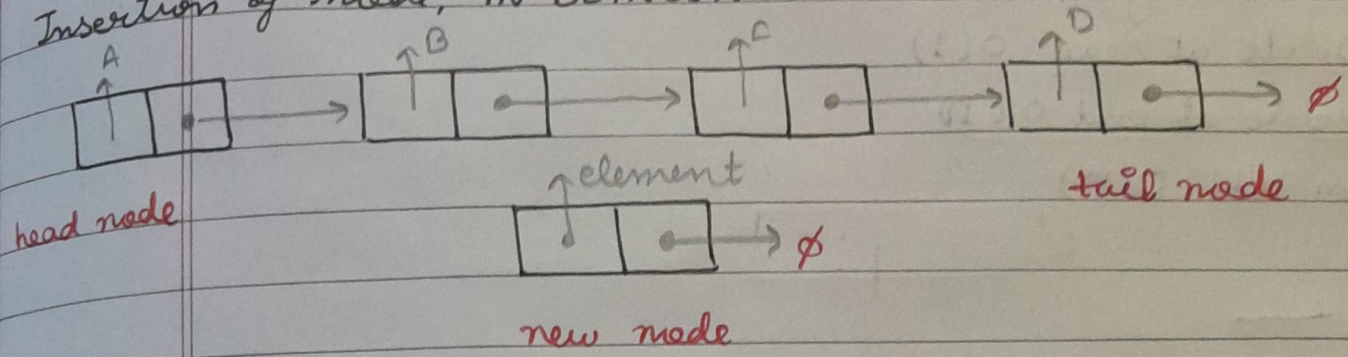
① Create a new node
② point new node.next from ∅ to head node
③ reassign the newnode as the head node
④ increases size = size + 1

Insertion of a node, AT TAIL



① Create new node
② point ~~new node~~ tail.next to ~~tail~~ new node
③ reassign new node as tail node. Increase ~~size~~ = size + 1

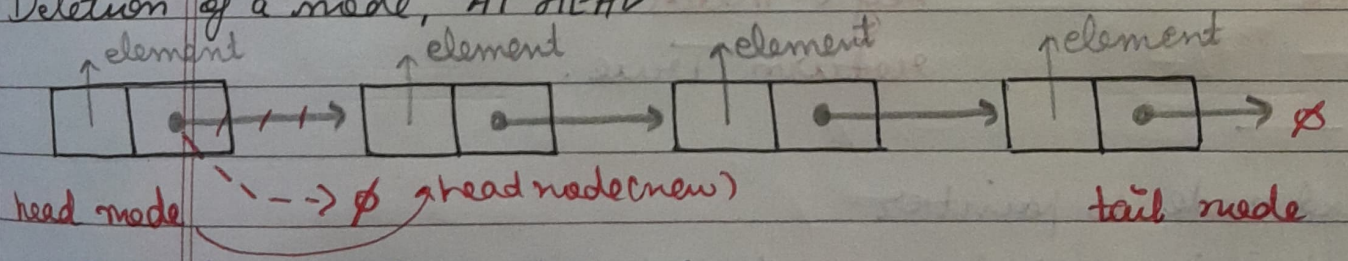
Insertion of node, IN BETWEEN



① Create a new node
② If you want it b/w B and C-
③ point B.next to new node
④ point new node.next to —? This is where it fails. As soon as we disconnect the list, we lose the other half. → Solved by Doubly Linked List.

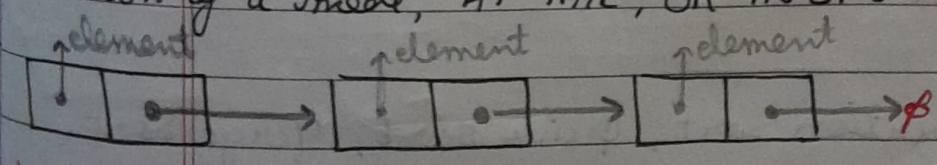
Deletion ——

Deletion of a node, AT HEAD



① Assign head.next to ∅
② Assign the next node as head node (new)
③ Decrease size by 1. size = size - 1.

Deletion of a node, AT TAIL, OR IN B/W. as soon as we disconnect, we lose the other half.



Linked list fails here, because no previous reference is found. Either break or go back.

oh we are last.
that means prev. should be II last.
How do we go back?

Linked list Indexing O(n)

LL Insert / Delete at head O(1)

LL Insert / Delete at end O(n)

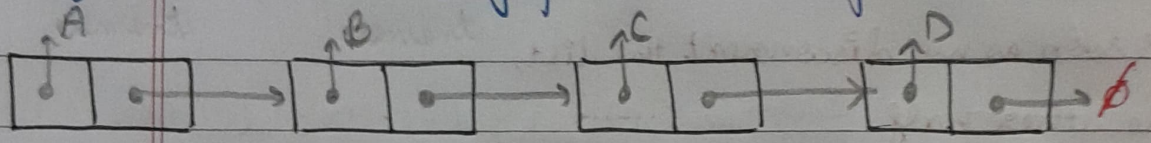
LL Insert in b/w O(n)

Searching —

① Make head as current node

② Run a loop till current node is null

③ In each iteration, match key of current node with item to find. Return true if found. Return false otherwise.



head node — CN i=2 → CN i=3 — tail node

CURRENT NODE in i=1

Key 'C' match return True.

Traverse —

① Set head node as pointer

② If pointer is null then empty list

③ If pointer != null, move pointer to wherever head.next point

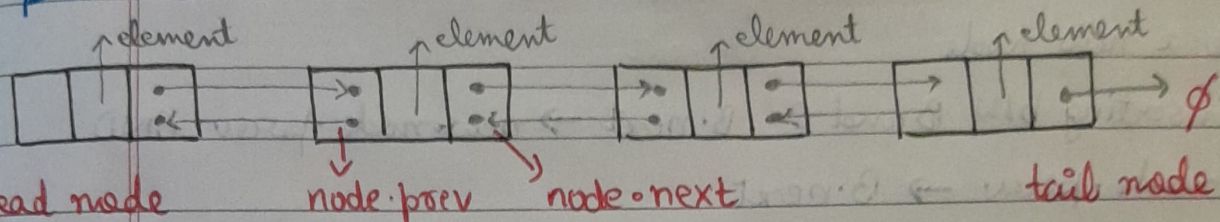
④ Repeat till pointer reaches null.

If for every iteration, you make i = i+1 we will get the size of the linked list.

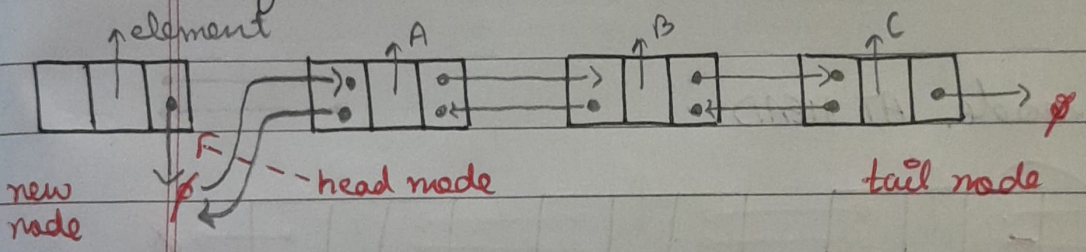
Sorting — Bubble sort, Insertion sort, Quick sort, Merge Sort.

DOUBLE LINKED LIST

Doubly linked list is a collection of nodes that forms a linear sequence. However, each node references to its element object, to the next node of the list, as well as to the next node of the previous node.



Insertion of a node, AT HEAD



① Create a new node

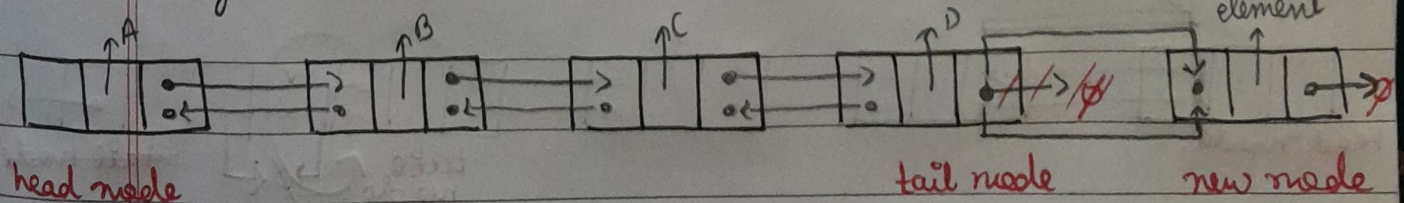
② Point new node.next → head node.previous

③ Point head node.prev → new node.next

④ Reassign the new node as the head node

⑤ Increase size = size + 1.

Insertion of a node, A TAIL

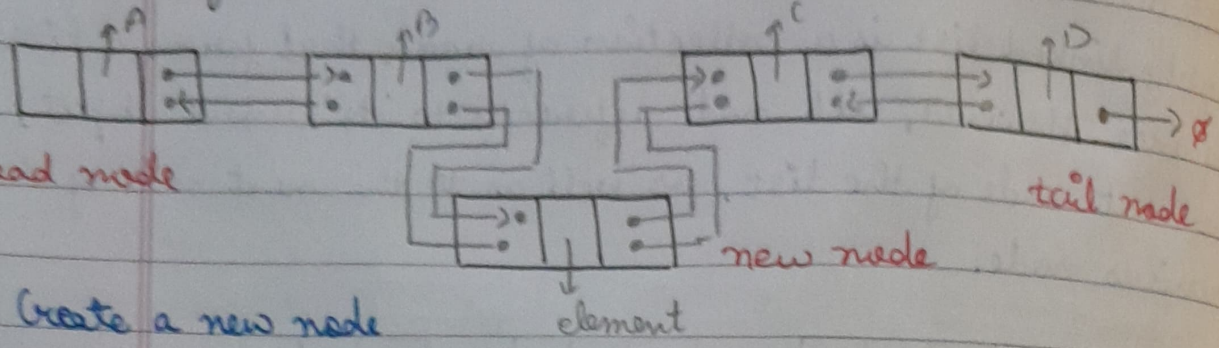


① ✓ ② Point new node.prev → tail node.next ③ tail node.next → new node.prev

④ Reassign new node as tail node ⑤ Increase size = size + 1

⑥ Point new node.next (now tail node.next) to ∅

Insertion of a node, IN BETWEEN



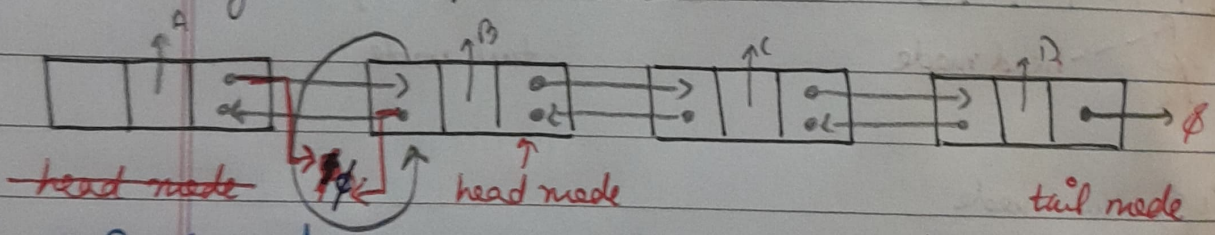
① Create a new node

② For inserting b/w B and C - B.next → new node.prev AND new node.prev → B.next

③ new node.next → C.prev and C.prev → new node.next

④ Increase size; size = size + 1.

Deletion of a node, AT HEAD



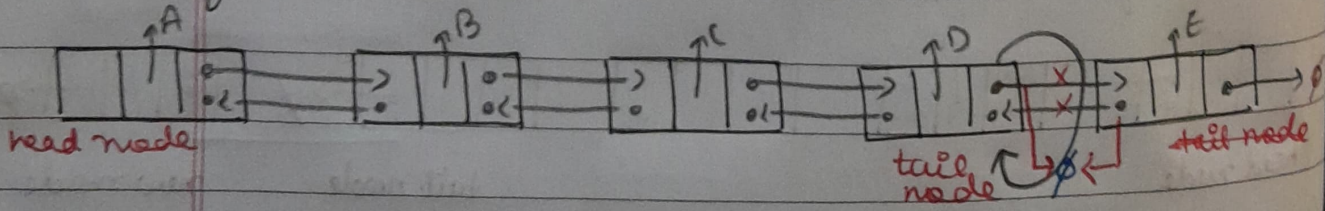
① Point headnode.next → ∅

② Point second node.prev → ∅ (break second to first link) → reset link

③ Reassign second node as head node

④ Decrease size, size = size - 1

Deletion of a node, AT TAIL



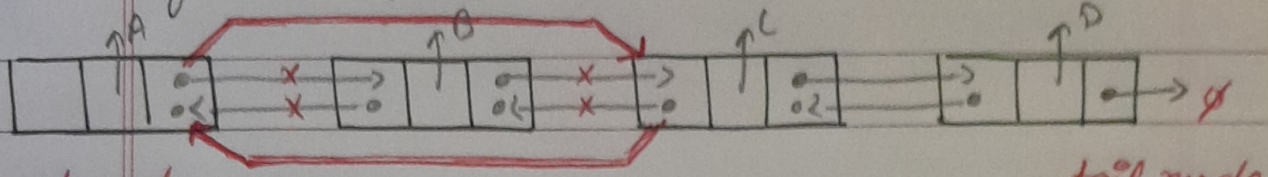
① tail.node.prev → ∅

② second last node.next → ∅

③ Reassign tail node

④ Size = size - 1 → reset link

Deletion of a node, IN B/W



① To delete B, A.next → C.previous

② C.previous → A.next

③ Decrease size = size - 1.

Complexity of insertion