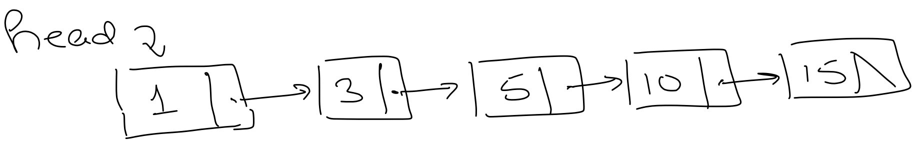
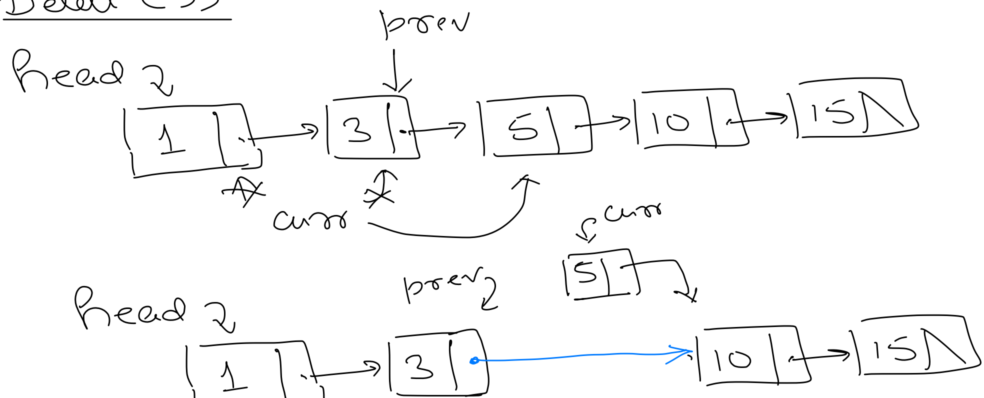
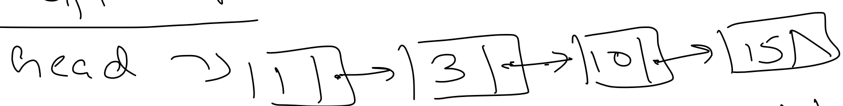
Delete a node from Singly linked list.



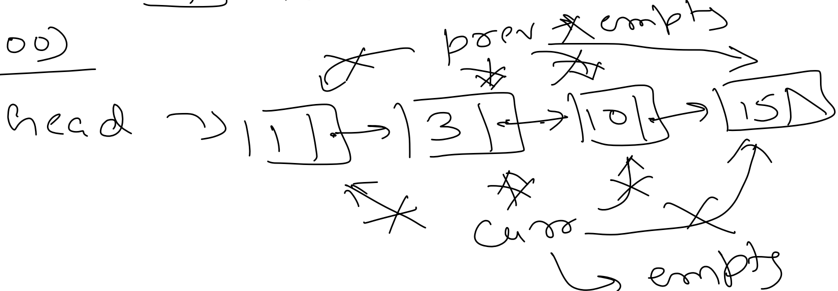
① Delete (5)



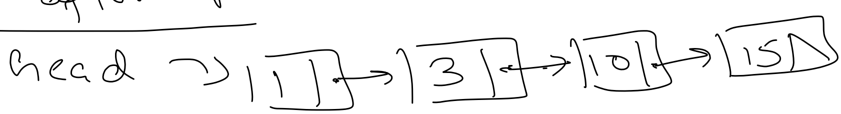
Result after op



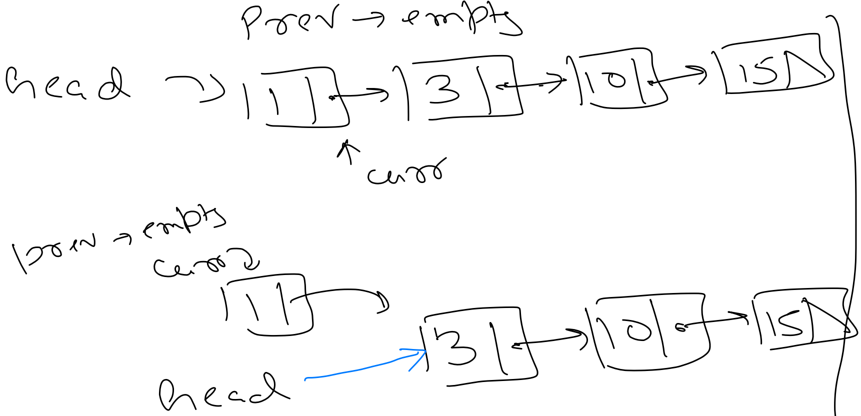
② Delete (100)



Result after op



③ Delete (1)



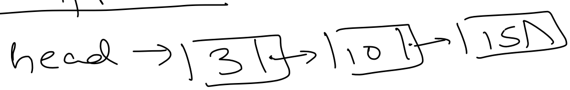
Delete (5)

head → empty

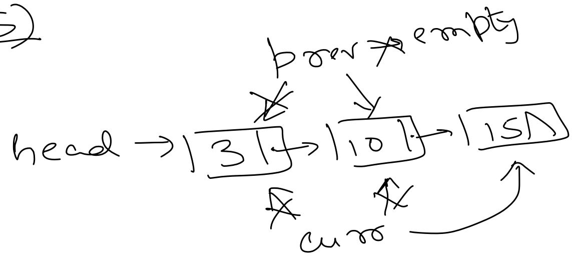
curr → empty

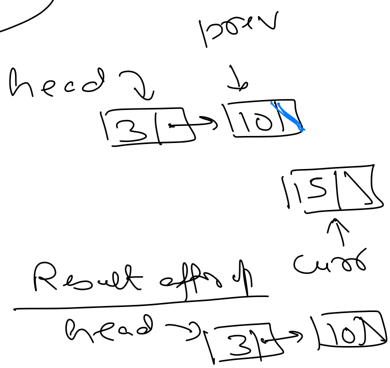
prev → empty

Result after op



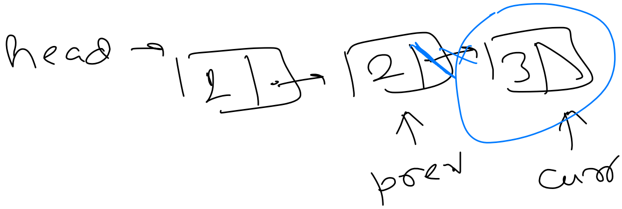
4) Delete (15)

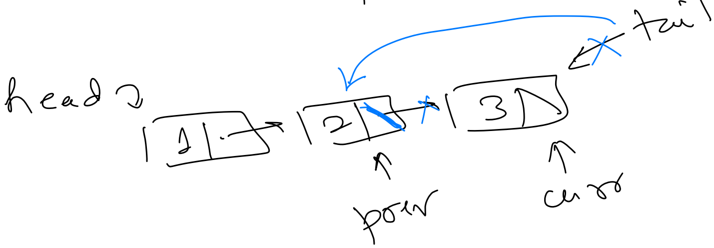


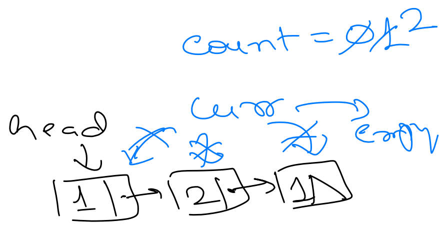


Delete(element)
- Set current to first node of list.
- Set previous to empty.
- while (current is not empty) do
  - if current node's data is element then
    - end the loop.
  - Set previous to current node.
  - Set current to current node's next.
- if current node is empty then
  // No node to be deleted as element not found OR list is empty.
  - Stop.
- if current node is the first node then
  // Deleting the first node of linked list.
  - Set head to current node's next.
  - Stop.
- Set previous node's next to current node's next.

Assignment: Modify Delete() algo for list that have tail ptr.







Exercises:

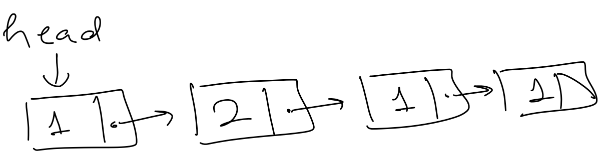
1. Count frequency of a given value.
   Int CountFrequency(int element);

CountFrequency(1)

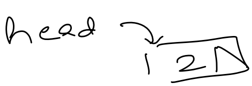
Test cases:
- Empty list.
- Frequency for element not present in list.
- Frequency of element present only once.
- Frequency of element present multiple times.

2. Delete all occurrences of a given number.

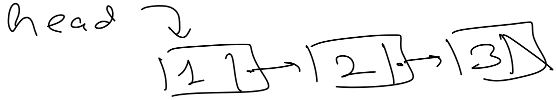
↳ 1. Find frequency of the number
2. Call Delete() frequency number of times.

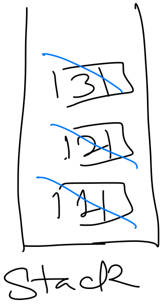


DeleteAll(1);



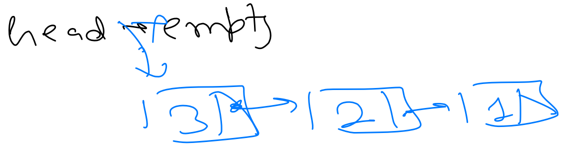
3. Reverse a singly linked list,



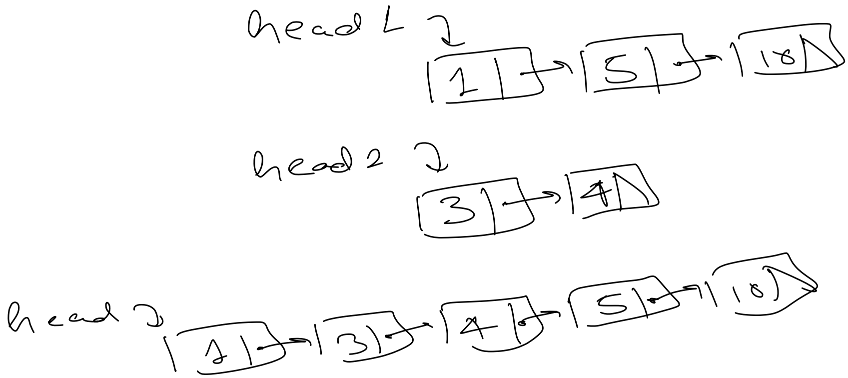


→ Traverse list & push nodes on stack.

→ Pop nodes from stack and append to list.



4. Merge two sorted linked list,

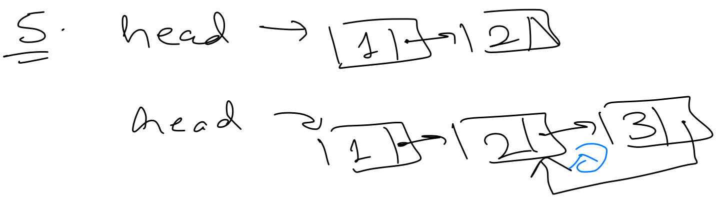


5. Finding cycle/loop in a linked list.
6. Finding intersection of two linked list.

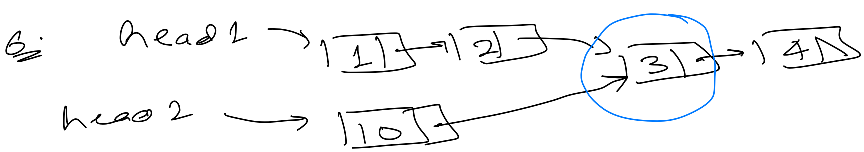
7. Insert element in sorted linked list using only current (no previous pointer to be used).

8. Delete node from a list using only current (no previous pointer to be used).

5.

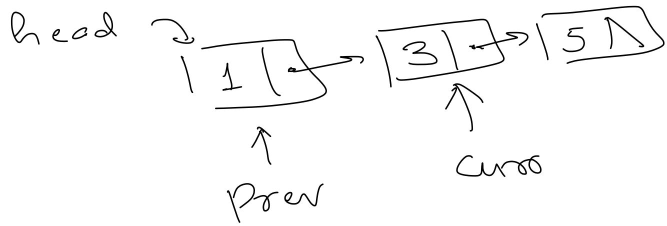


6.

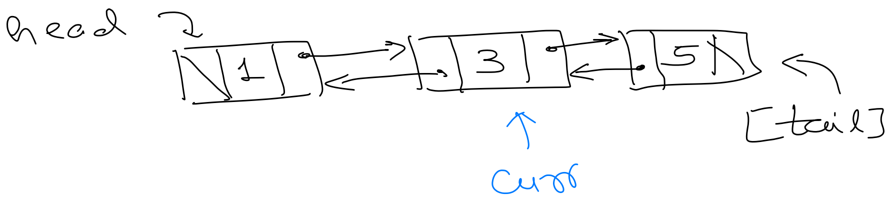


Doubly Linked List

→ Problem with Singly linked list



In singly linked list, each node keeps track of the node after it.
In doubly linked list, each node keeps track of the node after it & also of the node before it.



Doubly LinkedList Forward Traversal (Optimised)
- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node's next.

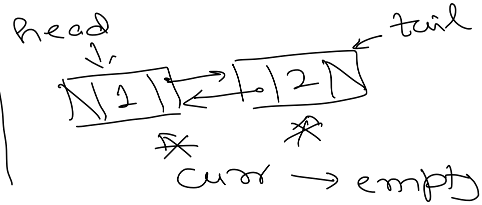
Traversal - Forward

① Traverse empty list

head → empty ← tail

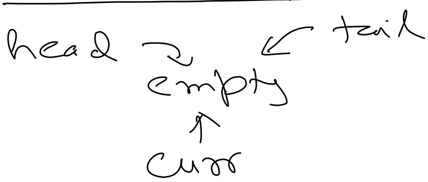
curr ↑

② Traverse non-empty list

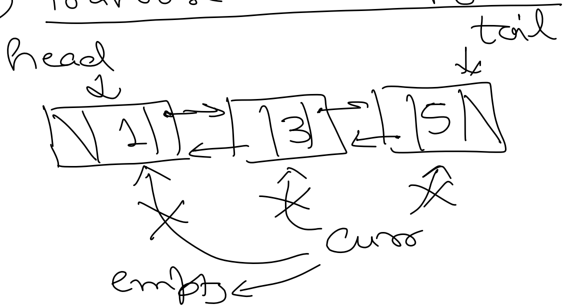


Traversal — Reverse

(1) Traverse empty list.



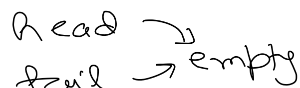
(2) Traverse non-empty list



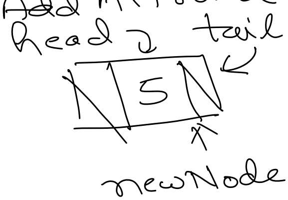
Doubly LinkedList Reverse/ Backward Traversal.
- Set current to last node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node's previous node.

AddAtFront(element) - Doubly Linked List
- Make space for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- Set newNode's previous to empty.
- if list is empty then
  - Set head and tail to newNode.
  - Stop.
- Set newNode's next to head.
- Set head node's previous to newNode.
- Set head to newNode.

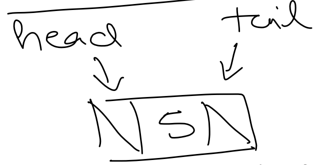
additional step to be done for doubly linked list



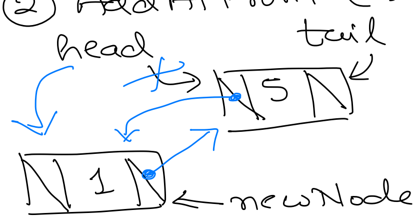
(1) ~~Add~~ AddAtFront(5)



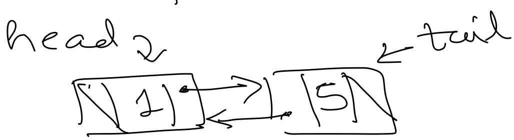
Result of op

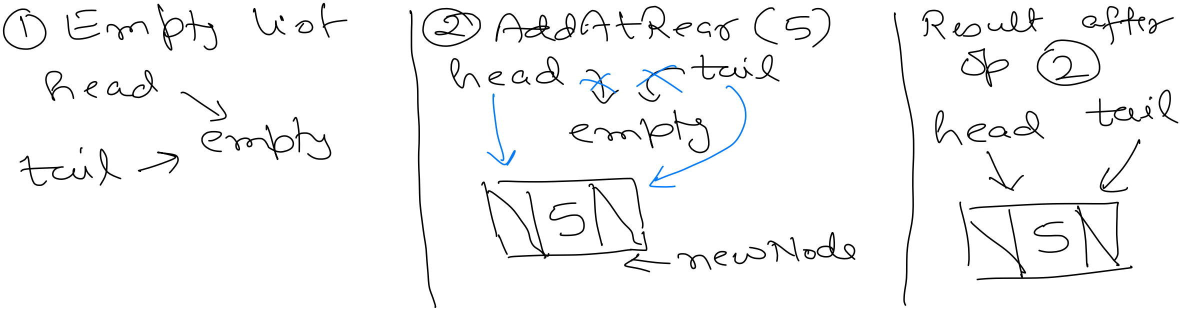


(2) AddAtFront (1)



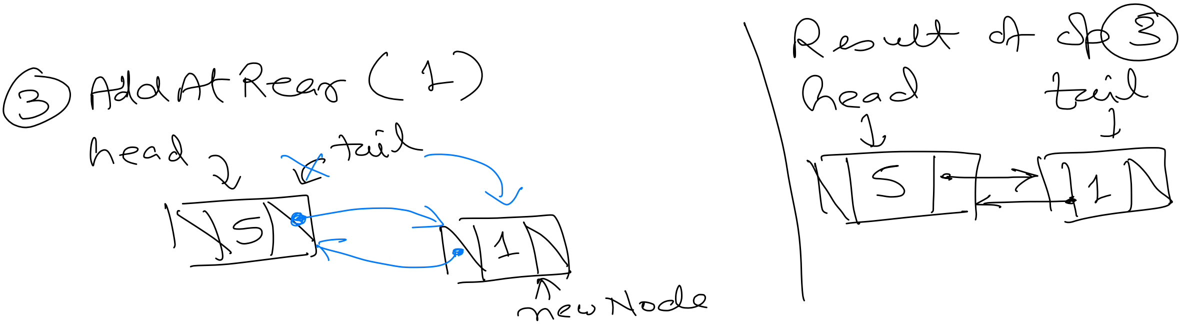
Result after op (2)

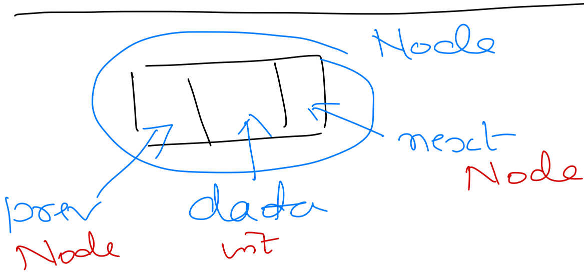




AddAtRear(element) - Doubly Linked List
- Make memory for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- Set newNode's previous to empty.
- if list is empty then
  - Set head and tail to newNode.
  - Stop.
- Set tail node's next to newNode.
- Set newNode's previous to tail.
- Set tail to newNode.

additional step to be done for doubly linked list






```
class Node {
    int data;
    Node next;
    Node prev;
}
```