Linear queue using array suffers from the problem that queue can be empty and full at same time.

0	1	2
~~5~~	~~10~~	~~20~~

front → ~~-1~~ ~~0~~ ~~1~~ 2

rear → ~~-1~~ ~~0~~ ~~1~~ 2

enqueue (5)

enqueue (10)

enqueue (20)

isFull() ⇒ TRUE

dequeue() ⇒ 5

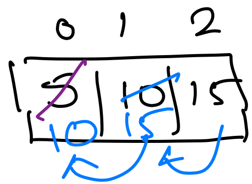
dequeue() ⇒ 10

dequeue() ⇒ 20

isEmpty() ⇒ TRUE

Solutions

(1) In dequeue(), after removing front element, shift all remaining elements to left by one place.



front → ~~-1~~ ~~0~~ -1

rear → ~~2~~ 1

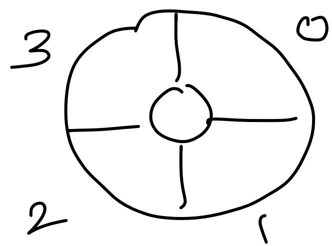
dequeue() => 5

(2) In dequeue(), after removing an element, we check if queue is empty and full at same time, if yes we reset front & rear.

(3) Implement circular queue.

Circular Queue

- Last position of Circular Queue is connected back to first position. Making a circle.



front → 0

rear → ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ → 0

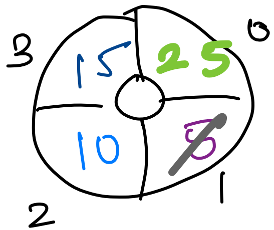
N = 4

0 . . (n - 1)

```
rear = rear +1;
 if (rear == N)
     rear = 0;
```

Incrementing of front and rear is a MOD N operation.

```
rear = (rear +1) % N;
```



N = 4

front = ~~0~~ 1

rear = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 0

enqueue(5)

enqueue(10)

enqueue(15)

~~enqueue(20)~~ throw exception Queue full

dequeue() ⇒ 5

enqueue(25)

isFull()

⇓

if front comes after rear.

$$(rear+1)\%N == front$$

N = queueData.length

Enqueue(element)
- If queue is full then stop.
- Make space at rear for new element. → rear = (rear + 1) % N
- Store new element and make it the rear element.

Dequeue()
- If queue is empty then stop.
- Move the front towards rear. → front = (front + 1) % N
- Remove the front element as result.
- Return result.

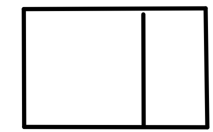
IsEmpty()
- If no elements stored in queue then return true.
Else return false.

IsFull()
- If no space left for new element to be stored then return true. → if ((rear + 1) % N == front)
return true;
Else return false.

Application of queue

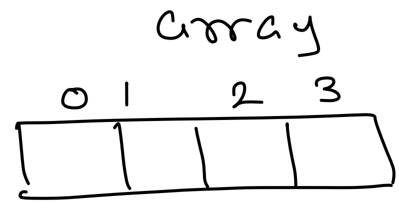
1. O.S. ⇒ Scheduler.
2. Simulation.
3. Other algorithms.

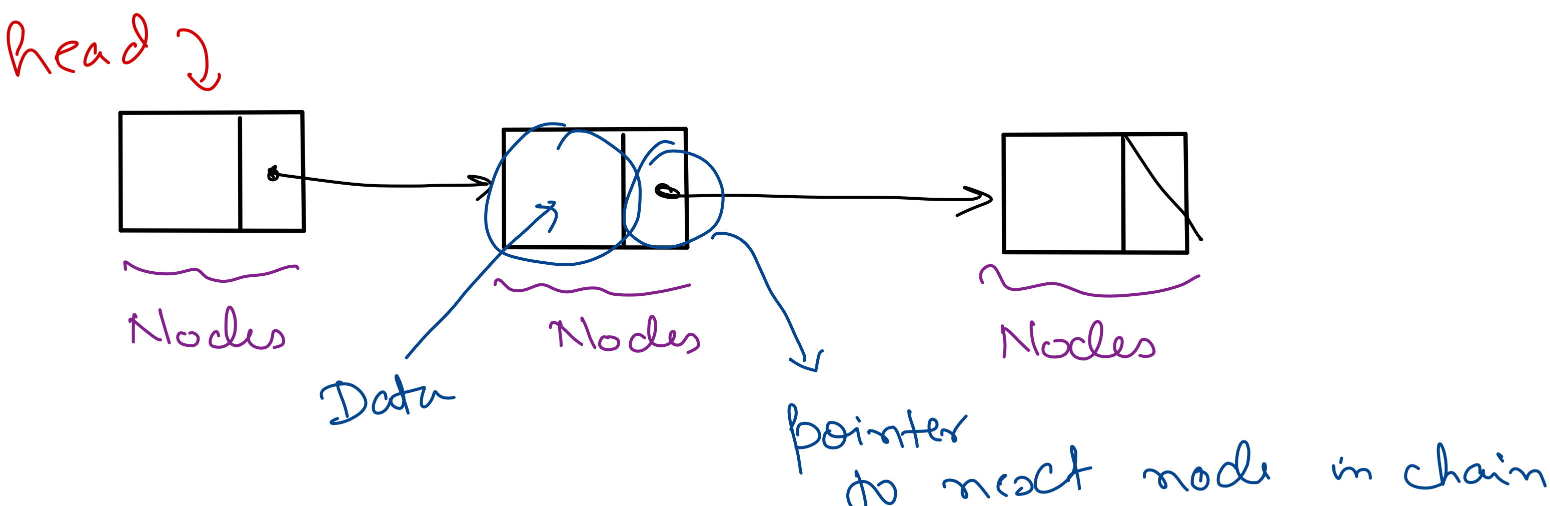
Linear Data Structures



Linked List

- Need for a linked list?





Properties of Linked List

- Stores data as a chain of nodes.

- Each node contains data and a pointer to the next node in the chain.

- First node of linked list is pointed by “head”.
  When list is empty, head do not point to any node.

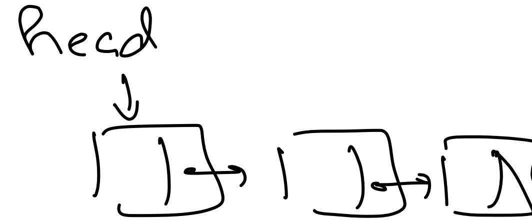
- Last node of list points to no node.

Pros and Cons of Linked List

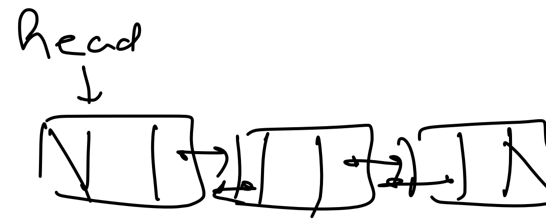
- Advantages
  - Can dynamically grow or shrink is size.
  - Efficient in insertion and deletion of elements.

- Disadvantages
  - Lookup OR Random access is inefficient.

Types of Linked List



- Single linked list (Uni-directional).
  One node keeps track of one neighbour node only.



- Doubly linked list (Bi-directional).
  Each node keeps track of two of its neighbours.

- Circular linked list.

Singly Linked List

Traversal

Starting from first element, access each element one at a time, till the last element.

Array Traversal

for i = 0 to (n-1)

... arr[i] ...

linked list traversal

① Empty list

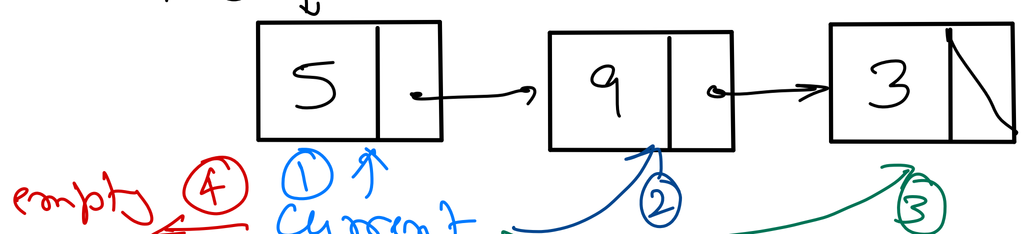
head → empty

list is empty, do nothing.

② Non-empty list

← non-empty list.

head



Singly LinkedList Traversal

- If list is empty then stop.
- Set current to first node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node's next.
- Stop.

Read → empty

Singly LinkedList 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.
- Stop.

Create Linked List

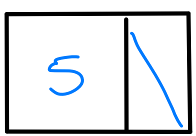
Add At Front

Initially, list will be empty. ⇒ new element will be the only element of list

addAtFront(5)

head → ~~x~~ empty

tail ~~x~~ empty

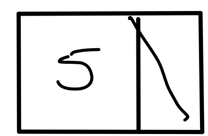


← new Node

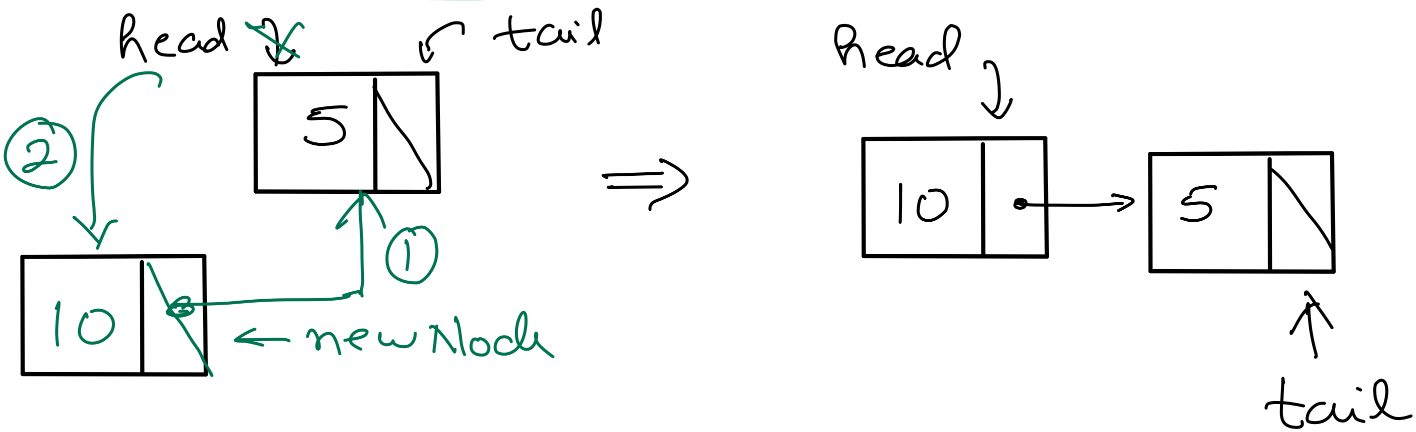
⇒

head

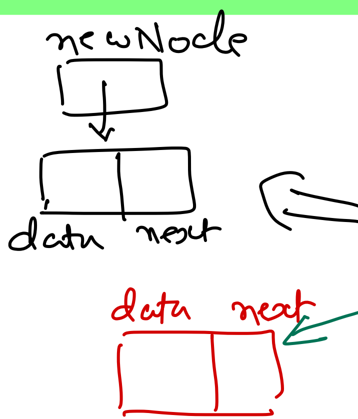
tail



addAtFront(10)




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



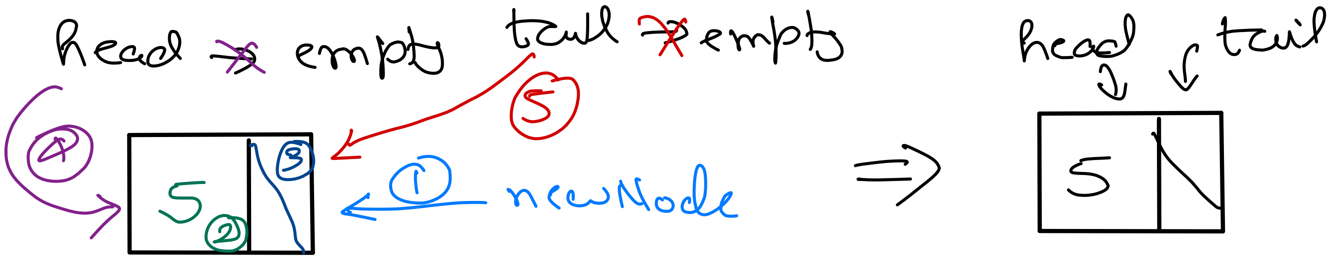
Node newNode;

newNode = new Node;

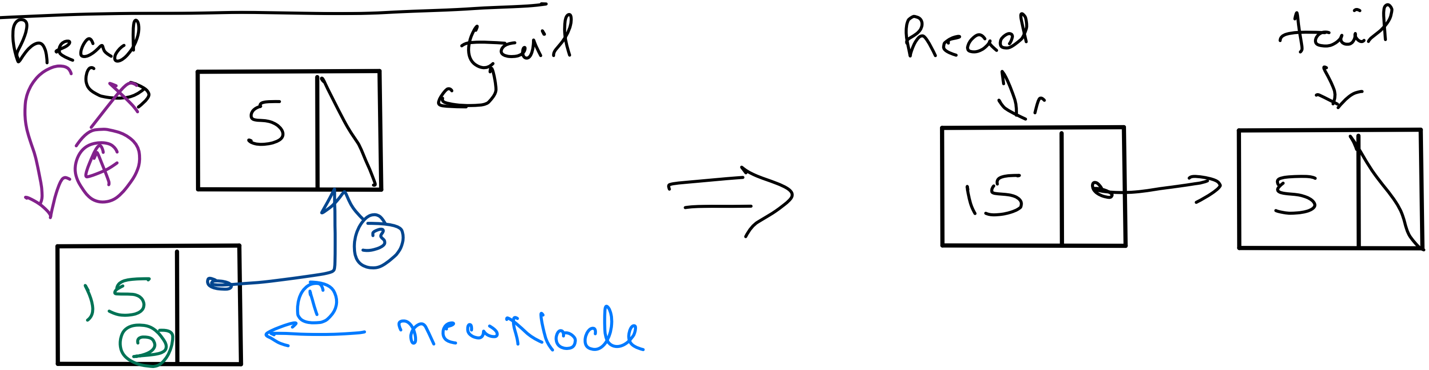
AddAtFront(element)

- Make space for new element, say newNode. → Node newNode = new Node;
- Store element in newNode's data. → newNode.data = element;
- Set newNode's next to empty. → newNode.next = null;
- if list is empty then → if (head == null) {
    - Set head and tail to newNode. → head = newNode; tail = newNode;
    - Stop. → return;
  }
- Set newNode's next to head. → newNode.next = head;
- Set head to newNode. → head = newNode;
- Stop.

addAtFront(5)



addAtFront(15)

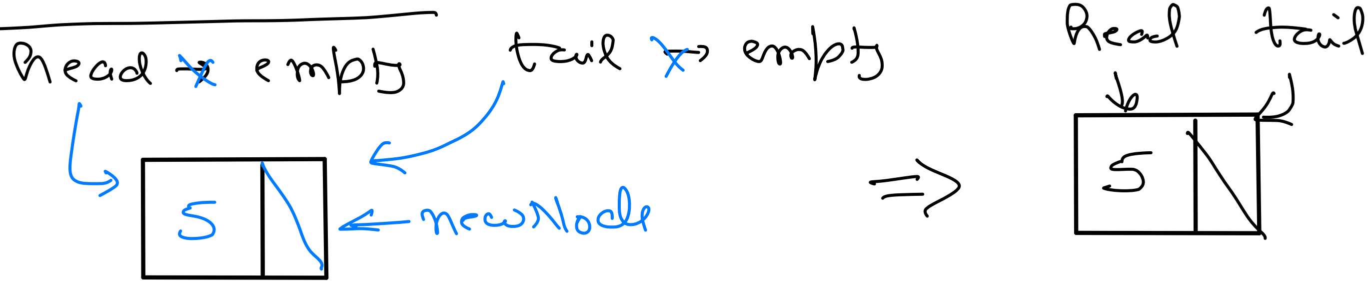


AddAtFront(element) - Optimised

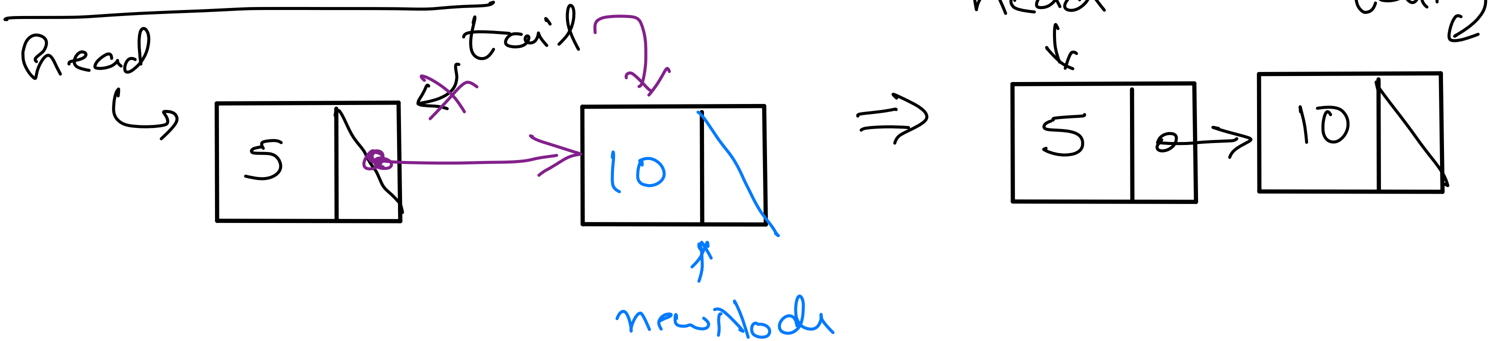
- Make space for new elements, say newNode. 1 Node newNode = new Node;
- Store element in newNode's data. 2 newNode.data = element;
- Set newNode's next to head. 3 newNode.next = head;
- Set head to newNode. 4 head = newNode;
- if tail is empty then
  - Set tail to head. 5 if (tail == null)
    tail = newNode;
- Stop.

Add At Rear / End ⇐ add newNode after last node of list.

addAtRear(5)



addAtRear(10)



AddAtRear(element)

- Make space for new elements, say newNode. → Node newNode = new Node;
- Store element in newNode's data. → newNode.data = element;
- Set newNode's next to empty. → newNode.next = null;
- if list is empty then → if (head == null) {
  - Set head and tail to newNode. → head = newNode; tail = newNode;
  - Stop. → return; }
- Set tail's next to newNode. → tail.next = newNode;
- Set tail to newNode. → tail = newNode;
- Stop.

Exercise: Implement addAtRear() without using tail (not keeping track of last node).

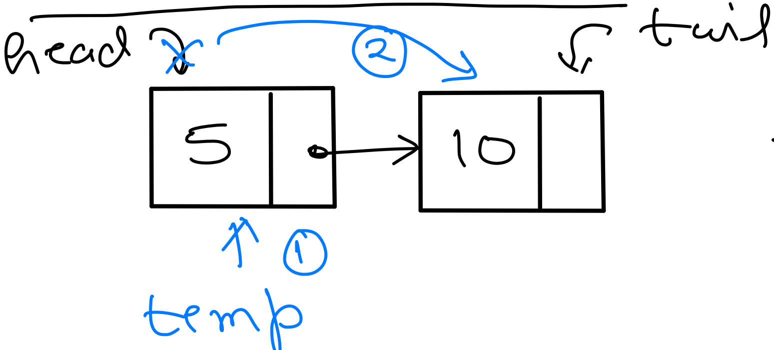
Hint: Traverse list to find last node.

Delete First Node

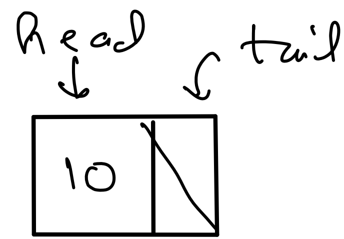
deleteFirstNode()

head → empty   tail → empty   ⇒ Stop as list is empty.

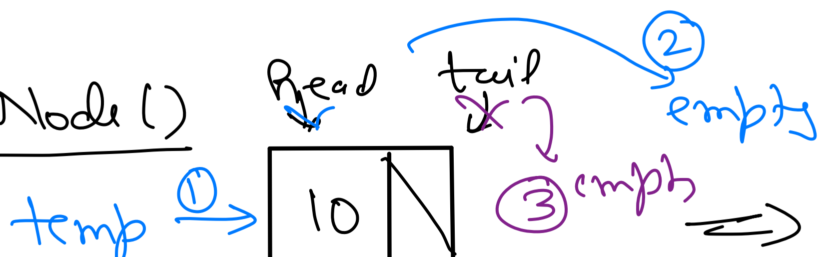
deleteFirstNode() ⇒ 5



⇒



deleteFirstNode()



⇒

head → empty

tail → empty.

DeleteFirstNode()

- if list is empty then
    - Stop
- Set temp to head.
- Set head to head's next.
- if list is empty then
    - Set tail to head.
- Stop.

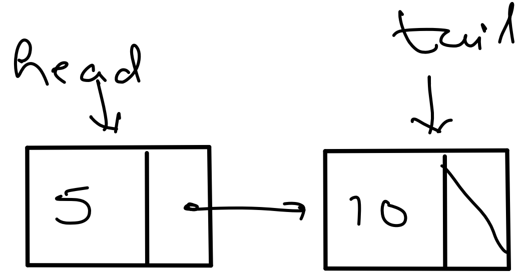
```
if (head == null)
    return;
temp = head;
 head = head.next;
if (head == null)
    tail = null;

return temp.data;
```

List as ADT

```
interface List {
        void addAtFront(int element);
        void addAtRear(int element);
        int deleteFirstNode();
        void print();
}
```

Implement Stack using linked list

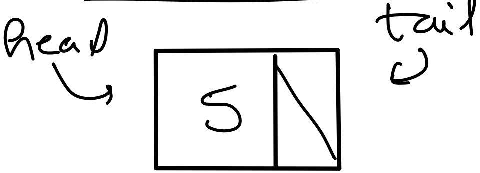
push()
↓
addAtFront()



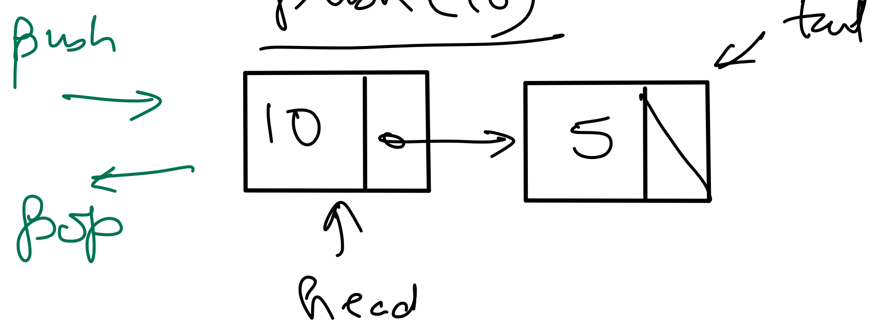
pop()
↓
deleteFirstNode()

---

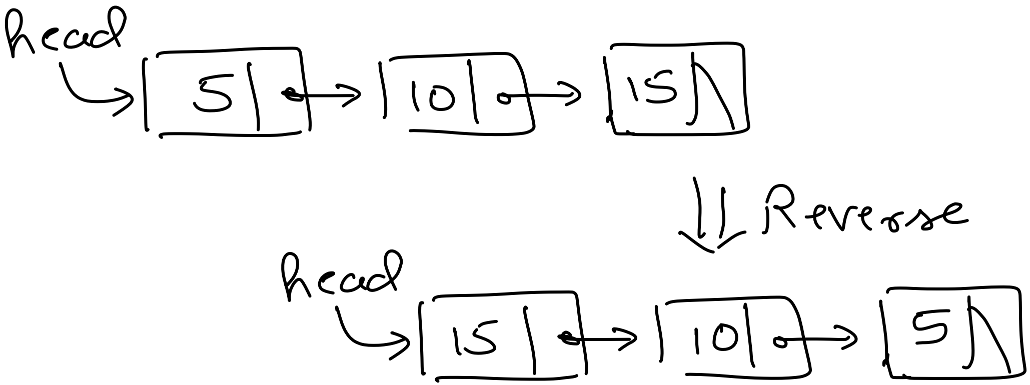
push(5)



push(10)



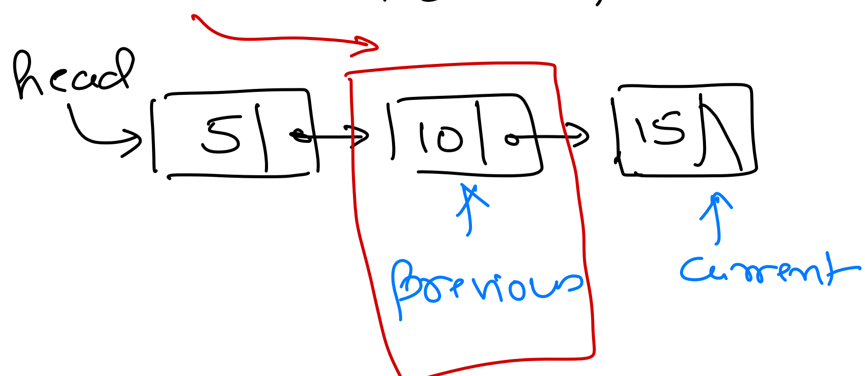
① Reverse a singly linked list.



→ Use stack

→ Use 3 pointers

② Find 2^{nd} last node of list.



→ Use stack

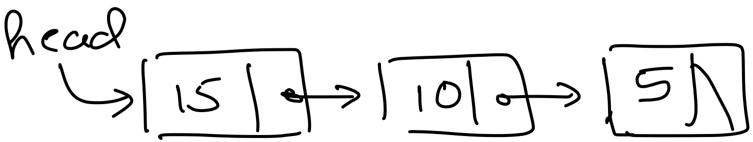
→ Use 2 pointers

3. Find kth last node of list.

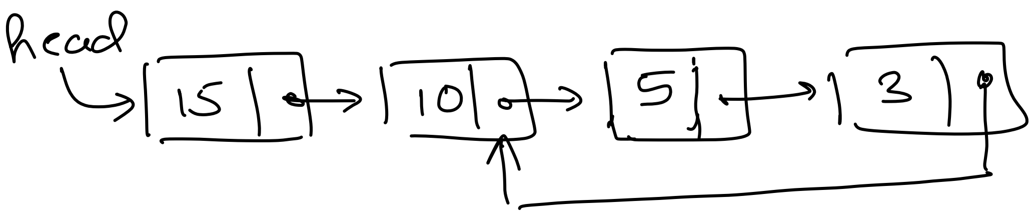
→ Use stack

→ Use 2 pointers

4. Detect if list contains a loop/cycle.



← No loop/cycle



← has loop/ cycle.

→ Keep track if a node is already visited or not.

→ Try reversing list.

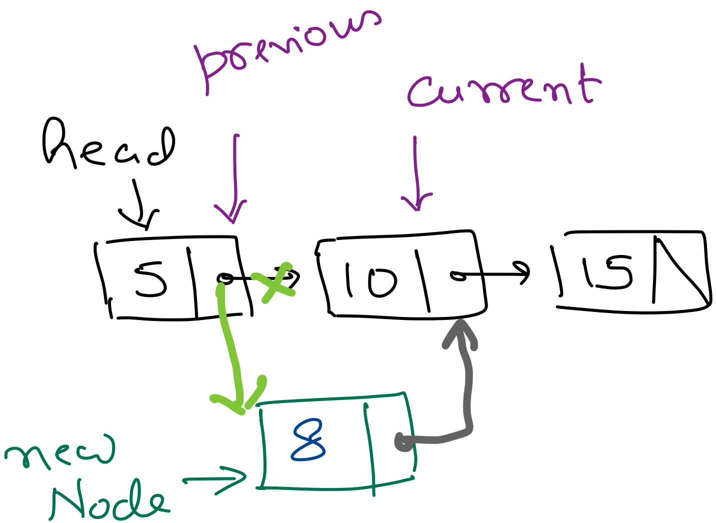
→ Two pointer (hare- tortoise)

fast - moves two nodes at a time.

slow - moves one node at a time

Insert in a list

- add element at a specific position.
- add element before/after a specific value.
- add element to a sorted list.



Step 1 : Create new Node.

Step 2 : Store element in newNode's data.

Step 3: Traverse list to find where new Node is to be added.

→ Set previous to empty
→ Set current to head.

→ while (current is not empty)

→ if (current node's data > newNode's data)

→ Node found, Stop.

→ Set previous to current.
→ Set current to current's next.

Step 4: Add newNode between previous and current.

1. Set previous node's next to newNode
2. Set newNode's next to current.

Special / corner cases

1. Empty list.
2. Adding smallest value to list.
3. Adding largest value to list.

```
Insert( element )
- Make space for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- if list is empty then
   - Make newNode as first (and only) node of list.
   - Stop

// List is not empty
// => Find first node having data greater than newNode's data.
- Set current to first node.
- Set previous to empty.
- while (current is not empty) do
   - if (current node's data > newNode's data) then
      // Found the node
      - End the traversal.
   - Set previous to current.
   - Move current to current's next node.
```

- if (previous is empty) then // newNode's data is smallest
  // Add newNode as first node.
  - Set newNode's next to first node.
  - Make newNode as first node.
  - Stop.

// Add newNode between previous and current
- Set newNode as next of previous.
- Set current as next of newNode.
- Stop.