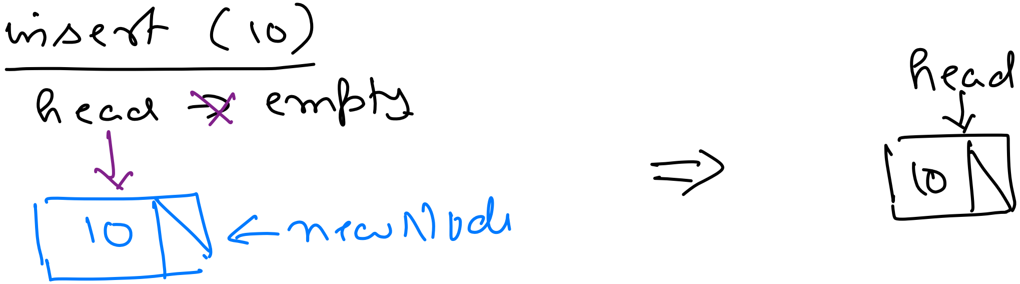
```
Insert( element )
- Make space for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- if list is empty then
   - Make newNode as first (and only) node of list.
   - Stop

// 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.
```

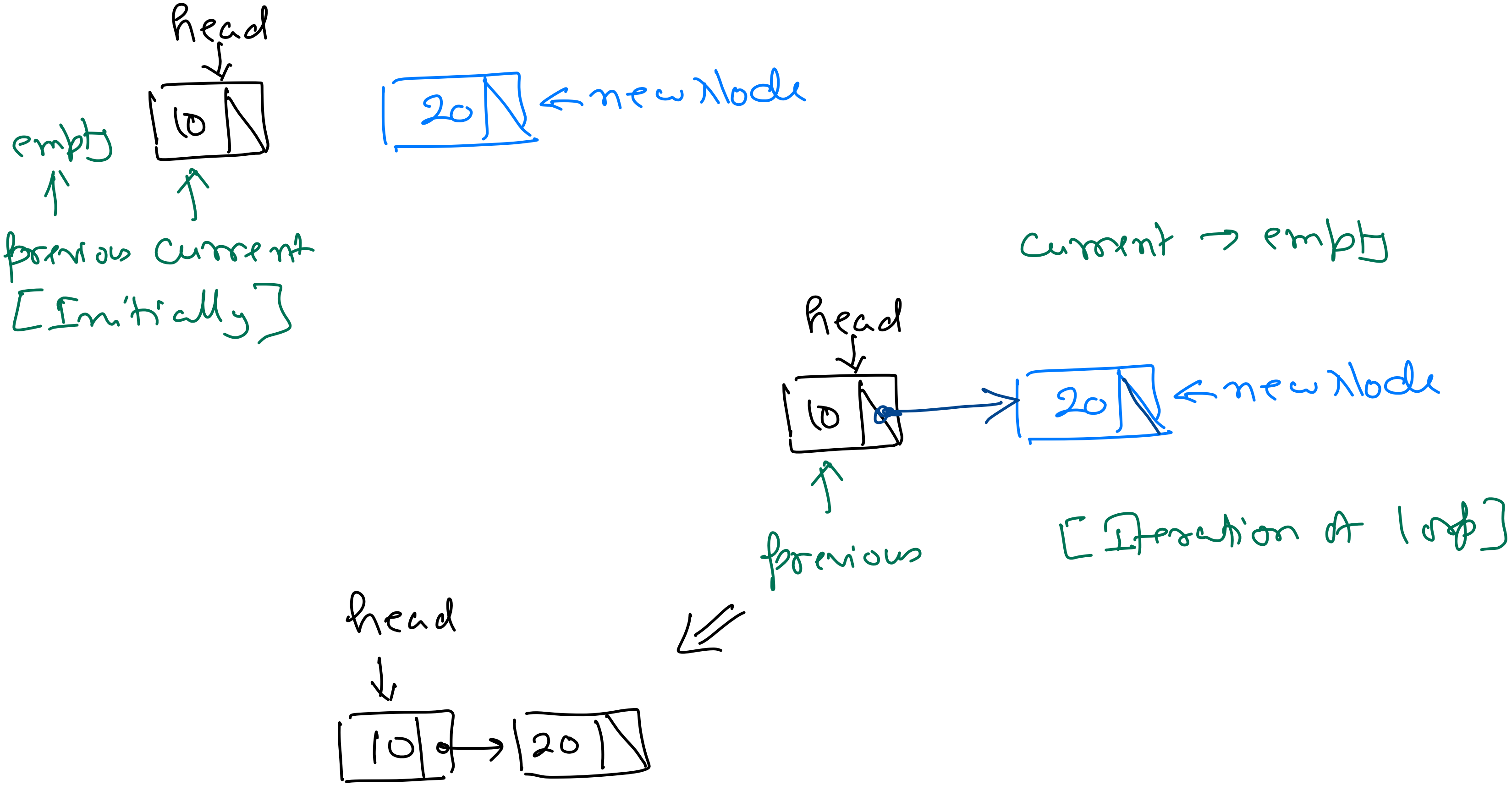
```
Node newNode =
      new Node(element);

if (head == null) {
      head = newNode;
      return;
}

current = head;
previous = null;
while (current != null) {
      if (current.data >
            newNode.data)
            break;
      previous = current
      current = current.next;
}
```

```
- 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.
```

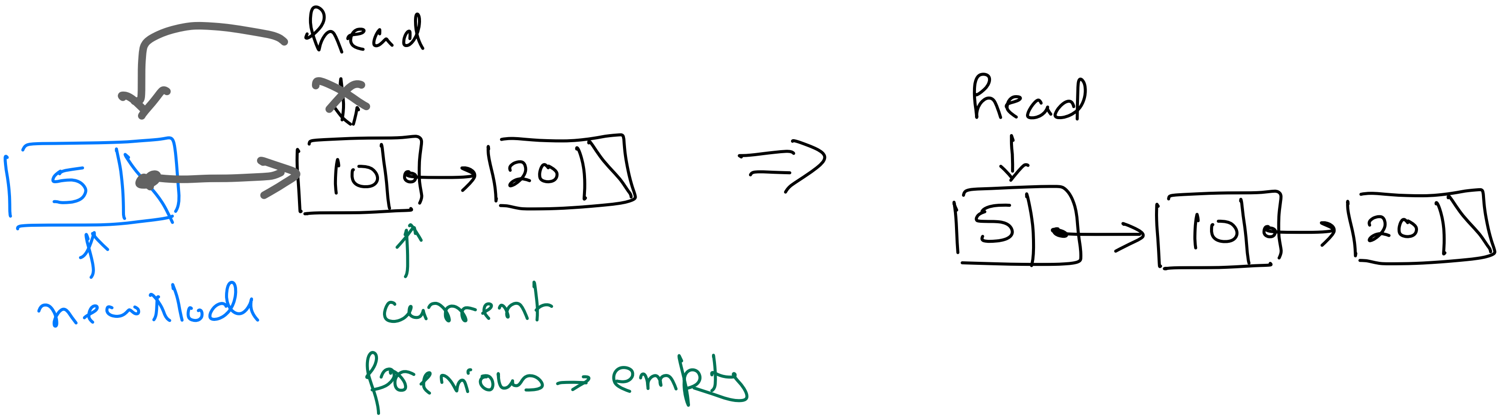
```
if (previous == null){
    newNode.next = head;
    head = newNode;
    return;
}
previous.next = newNode;
newNode.next = current;
```



insert (20)


head
10
20
← newNode
empty
previous
current
[Initially]
current → empty
head
10
20
← newNode
previous
[Iteration of loop]
head
10
20

insert (5)



Insert in a list using Single pointer (current).

① Do not let current become empty.

⇓

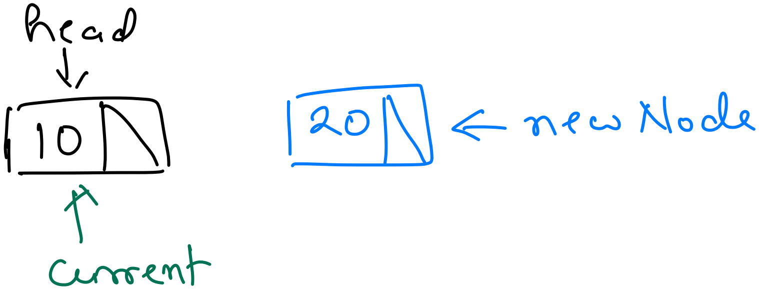
Stop traversal at last node.

while (current.next != null)

Add new Node AFTER current node.

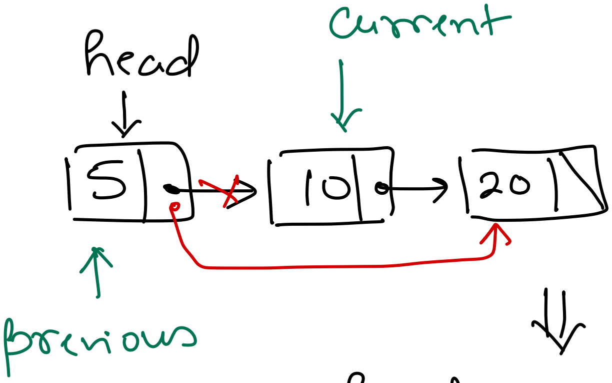
⇓

Check current's next node having data > newNode's data.



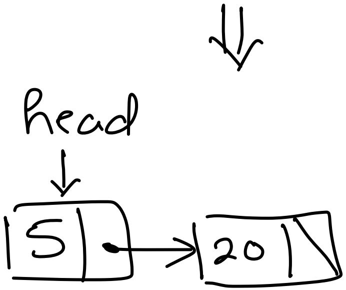
② After traversal, if current is last node then we need to again check if add before or after current node.

Delete a node



delete (10)

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

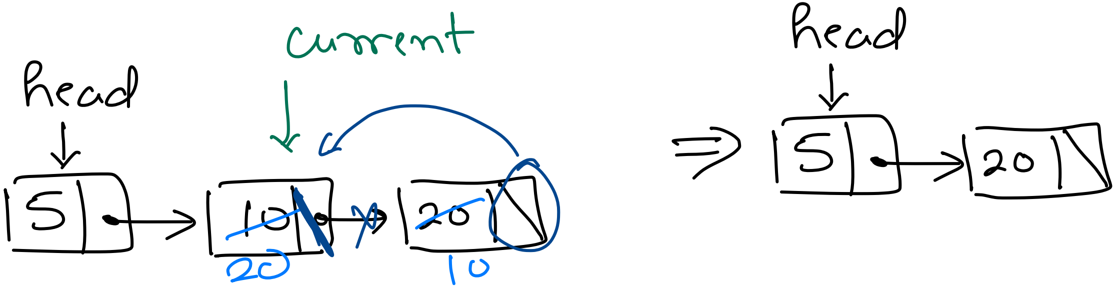


Special cases

1. Element not found.
2. Deleting first node.

```
Delete( element )
- Set current to first node (head)
- Set previous to empty.
- while (current is not empty) do
   - if (current node's data = element) then
      - End the traversal. // Element found
   - Set previous to current node.
   - Move current to current node's next node.

- if (current is empty) then // Element not present in list
   - Stop

- if (current node is first node of list) then // Deleting first node
   - Move head to head's next node.
   - Mark current node as free. ← Not needed in JAVA
   - Stop.

- Set previous node's next to current node's next.
- Mark current node as free. ←
- Stop.
```

Delete a given node



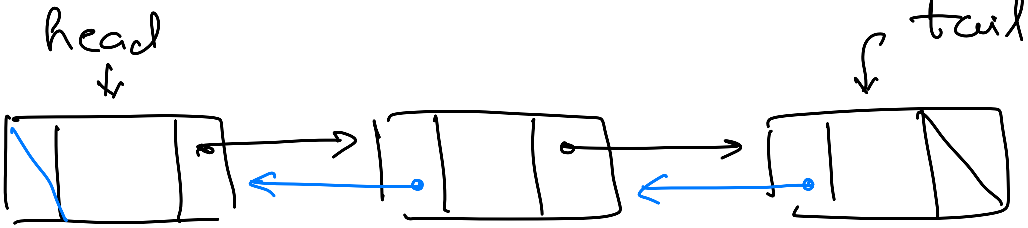
① Swap current and its next node's data

② Delete current's next node.

} will not work if current is last node.

Doubly Linked List

linked list in which each node keep track of its two adjacent nodes.



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

Traversal → Forward Traversal ⇒ Same as Singly list traversal.

→ Backward Traversal

⇓

Start with last node and move backwards

Forward Traversal

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

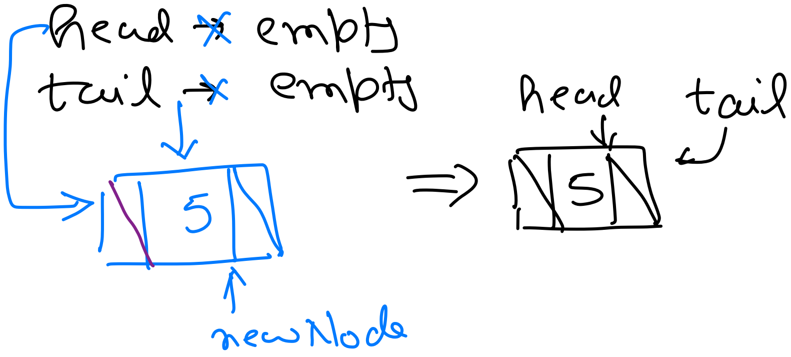
Backward Traversal

```
- Set current to last node of list.
- while (current is not empty) do
  - Process current node.
  - Set current to current node's previous .
- Stop.
```

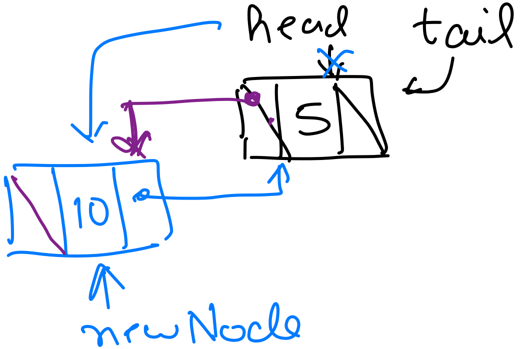
```
AddAtFront( element )
- Make space for new element, say newNode.
- Store element in newNode's data.
- Set previous of newNode as empty.
- if list is empty then
  - Make newNode as last node.
 Else
  - Set newNode as previous of head.
- Set newNode's next to head.
- Set head to newNode.
- Stop.
```

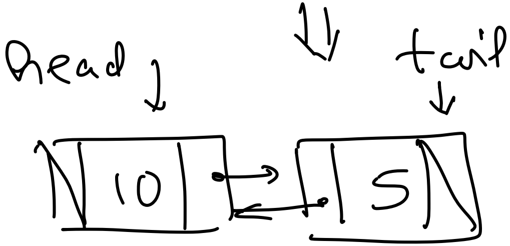
```
Node newNode = new Node(
                 element)
if (head == null) {
        tail = newNode;
} else {
    head.previous = newNode;
}
newNode.next = head;
head = newNode;
```

addAtFront (5)



addAtFront (10)

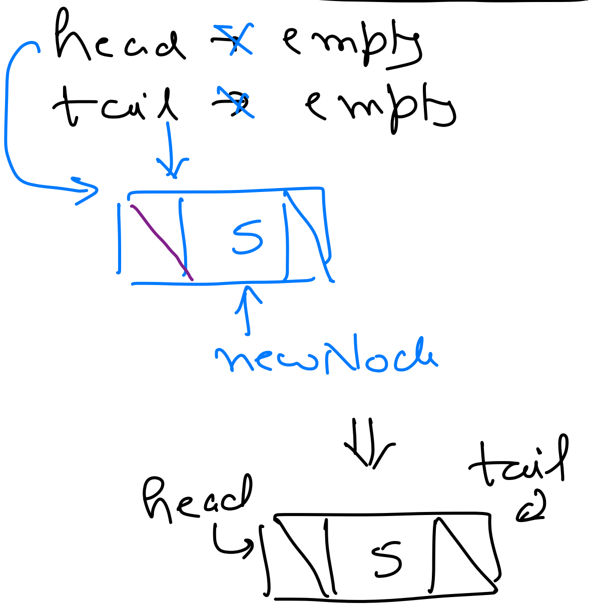




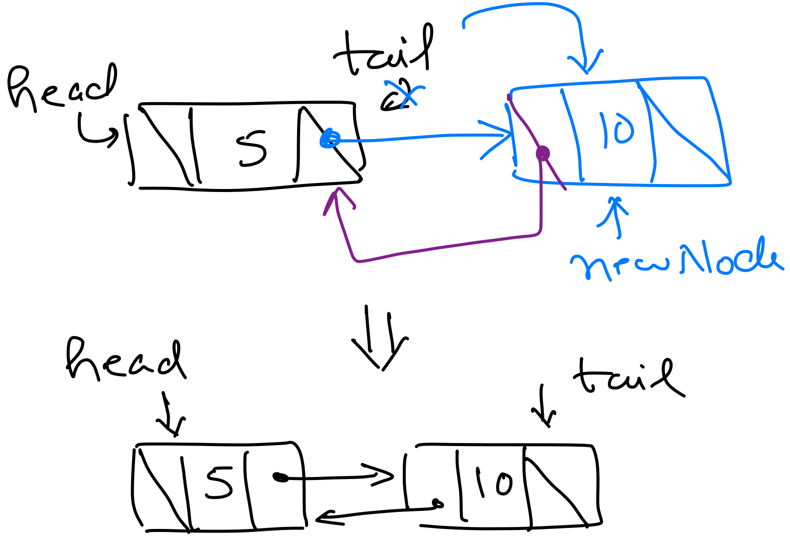
addAtRear (5)

head ~~=~~ empty

tail ~~=~~ empty

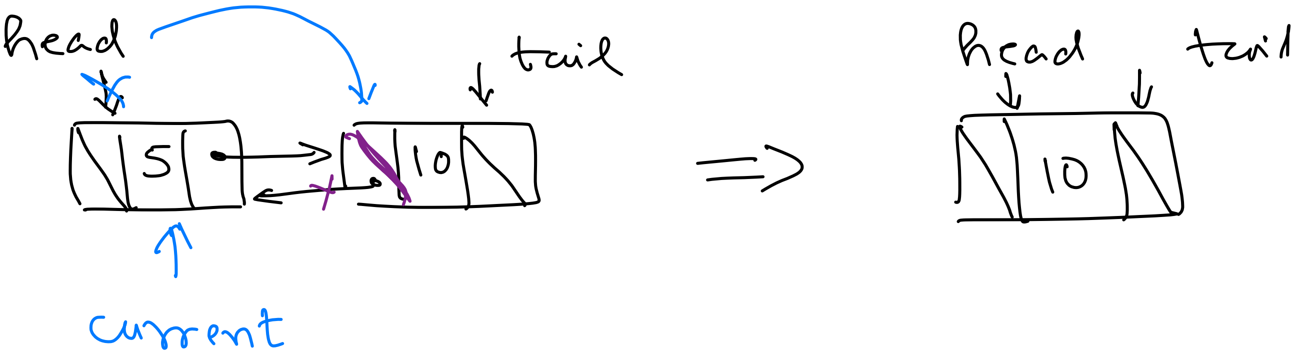


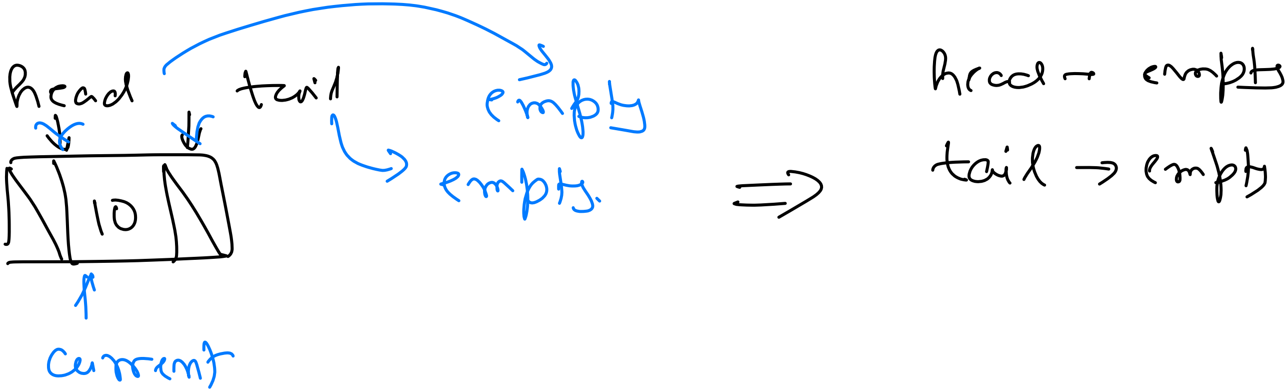
addAtRear (10)



```
AddAtEnd( element )
- Make space for new element, say newNode.
- Store element in newNode’s data.
- Set newNode’s next to empty.
- Set previous of newNode as empty.
- if list is empty then
    - Set head to newNode.
    - Set tail to newNode.
    - Stop.
- Set newNode as next of tail node.
- Set previous of newNode to tail.
- Set tail to newNode.
- Stop.
```

Delete First Node

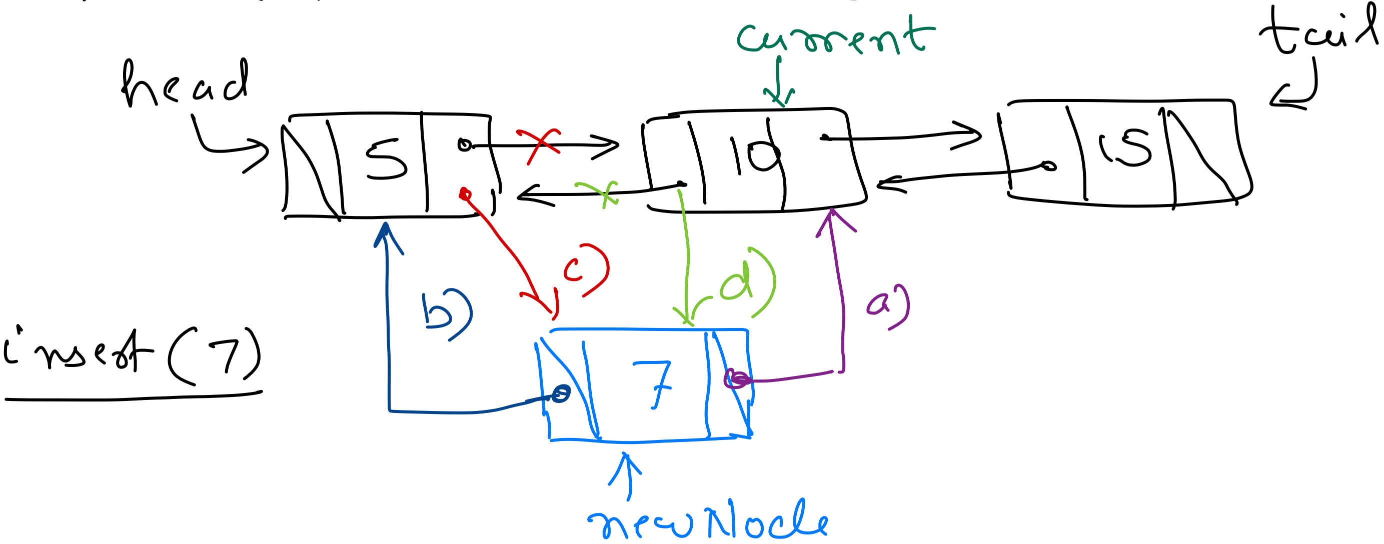




```
DeleteFirstNode()
- if list is empty then
    - Stop.
- Set current to head
- Move head to head's next node.
- if list is empty then // Check if head is empty, as head got changed
    - Set tail to empty
 Else // list is not empty
    - Set previous of head to empty
- Release current node. ← Not required in JAVA.
- Stop.
```

Add Element to doubly list in other ways

→ Create sorted doubly list.



a) Set newNode's next to current.

b) Set newNode's previous to current node's previous.

c) Set current node's previous node's next to newNode.

d) Set current node's previous to newNode.

Special cases

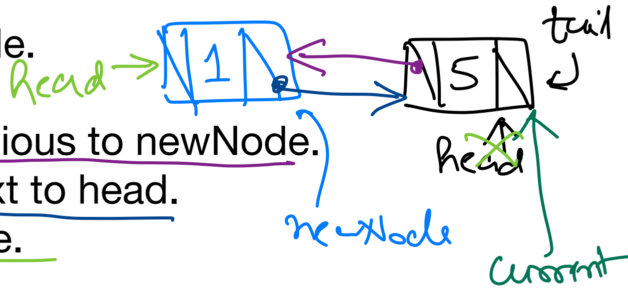
1. List is empty. ⇒ Set newNode as first and last node.

2. Adding smallest value. ⇒ add newNode before first node.

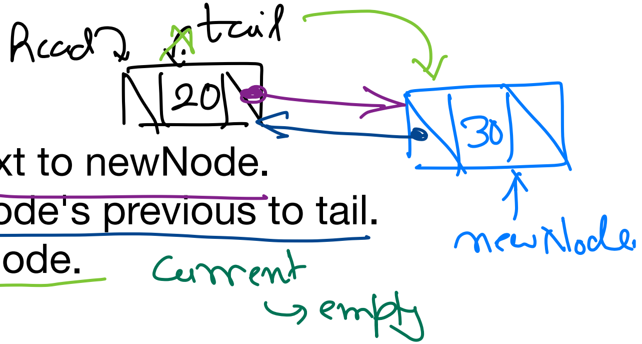
3. Adding largest value. ⇒ add newNode after last node.

```
Insert( element )
// 1. Create new node
- Make memory for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next and previous to empty.

// 2. If list is empty?
- if head is empty then
  // Make newNode as the first and last node of the list.
  - Set head and tail to newNode.
  - Stop.

// 3. Traverse list to find node - current node.
- Set current to head (first node).
- while (current is not empty) do
  - if (current node's data > element) then
     // Found the node, end the traversal.
     - End the traversal.
  - Set current to current's next node.
```

```
// 4. If adding before the first node? - Current is the first node.
- if (current is head) then
  - Before the first node comes newNode. // Set head's previous to newNode.
  - After newNode comes the first node. // Set newNode'next to head.
  - Make newNode as the first node. // Set head to newNode.
  - Stop.
```



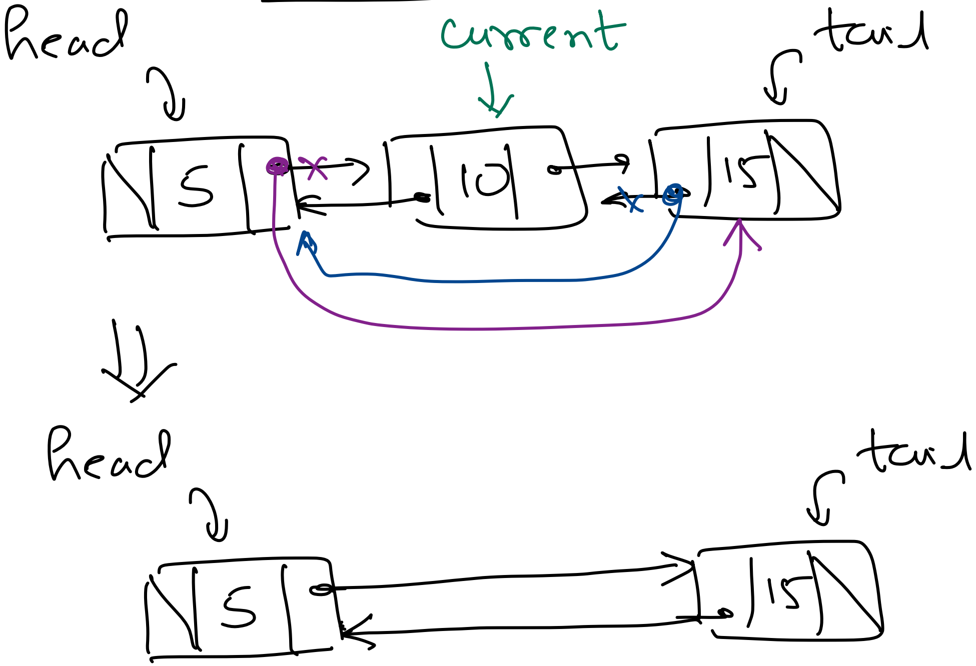
```
// 5. If adding after the last node? - Current is empty
- if (current is empty) then
  - After the last node comes newNode. // Set tail's next to newNode.
  - Before newNode comes the last node. // Set newNode's previous to tail.
  - Make newNode as the last node. // Set tail to newNode.
  - Stop.
```



```
// 6. Add a new node between current and current’s previous node.
- Make the current node come after newNode. // Set newNode's next to current.
- Make the current node's previous node come before newNode. // Set newNode's previous to current node’s previous.
- Make newNode come after the current node's previous node. // Set current node's previous node's next to newNode.
- Make newNode come before the current node. // Set current node's previous to newNode.
- Stop.
```

Delete node from doubly list

delete (10)



a) Set current node's previous node's next to current node's next.

b) Set current node's next node's previous to current node's previous.

Special cases

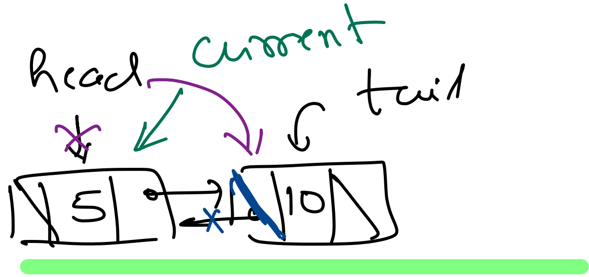
① list is empty. => Do nothing.

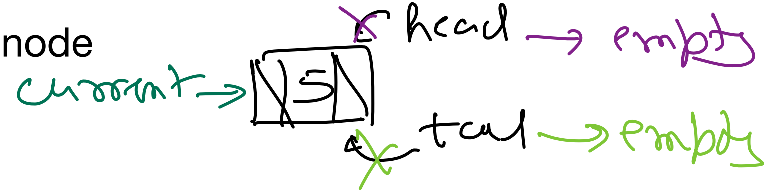
② Current is empty. => Do nothing.

③ Current is first node. => Delete first node.

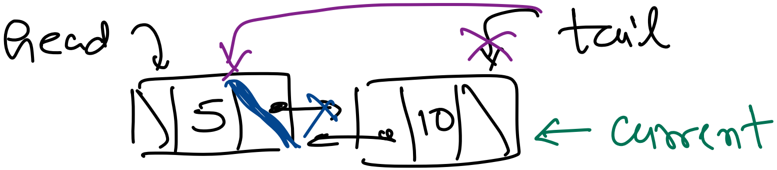
④ Current is last node. => Delete last node.

```
Delete (element)
// Find the node to be deleted - current node
- Set current to first node (head)
- while (current is not empty) do
  - if (current node's data = element) then
     // Found the node - end the traversal.
     - End the traversal.
  - Move current to current's next node

// Have we found the node to be deleted?
- if (current is empty) then
  - Stop.

// Delete first node?
- if (current is first node) then
  - Move head to head's next node.
  // Has the list become empty => list has only 1 node
  - if (head is empty) then
    - Set tail to empty.
  Else
    - Set the previous of head to empty.
  - Release memory of the current node. (Not required for JAVA).
```





```
  - Stop.

// Delete last node?
- if (current is last node) then
  - Move tail to tail’s previous node.
  - Set the next of tail node to empty.
  - Release memory of the current node. (Not required for JAVA).
  - Stop.
```

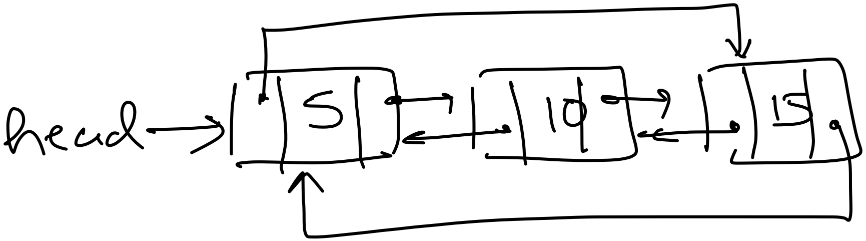


- Make current's next node come after current's previous node. // **Set current node's previous node's next to current node's next node.**
- Make the current node's previous node come before the current node's next node. // **Set current’s next node's previous to current’s previous node.**
- Release memory of the current node. (Not required for JAVA).
- Stop.

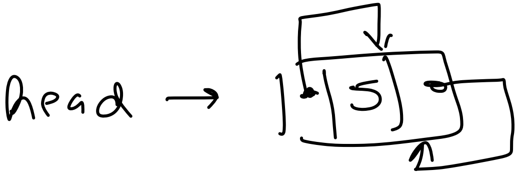
Circular Doubly Linked List

How to implement it?

Issue if first and last nodes are connected to form a cycle.
How will we do the traversal?



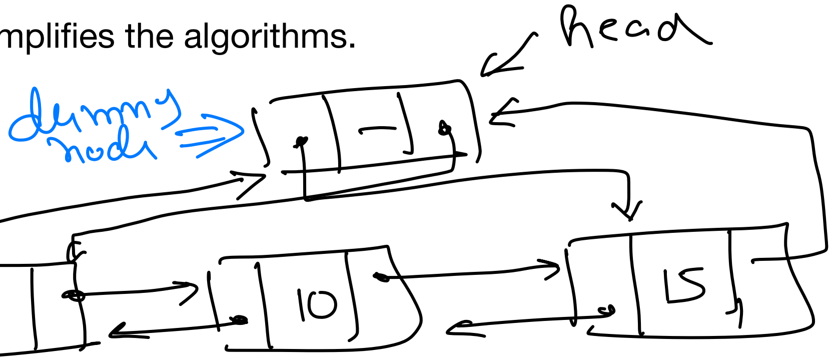
head → empty

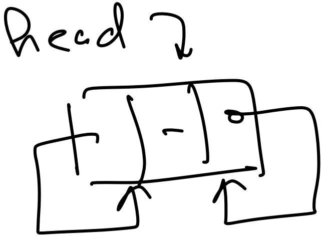


Traversal

```
if (head == null)
    return;
current = head;
while (current.next != head)
    process current;

    current = current.next;

}
process current.
```

How using a dummy node, simplifies the algorithms.





Empty circular list

Traversal

```
current = head.next;
while( current != head)
        process current;
    current = current.next;
}
```