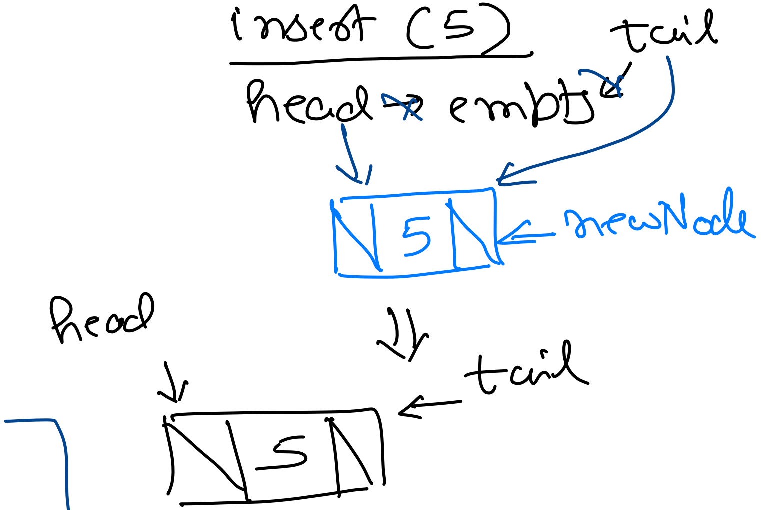
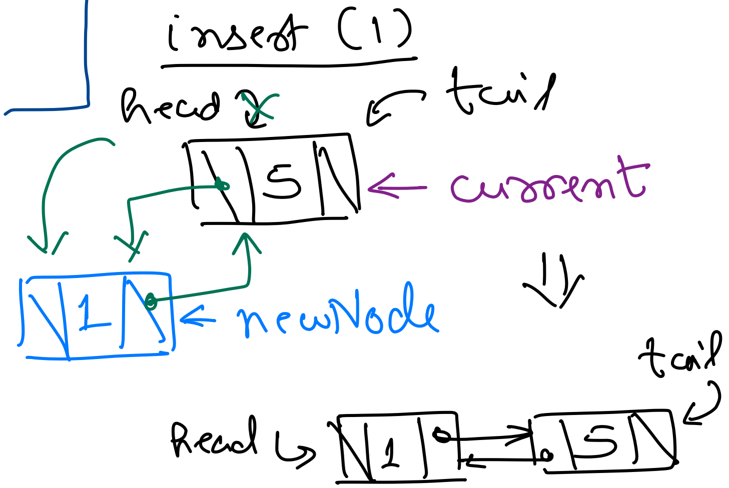
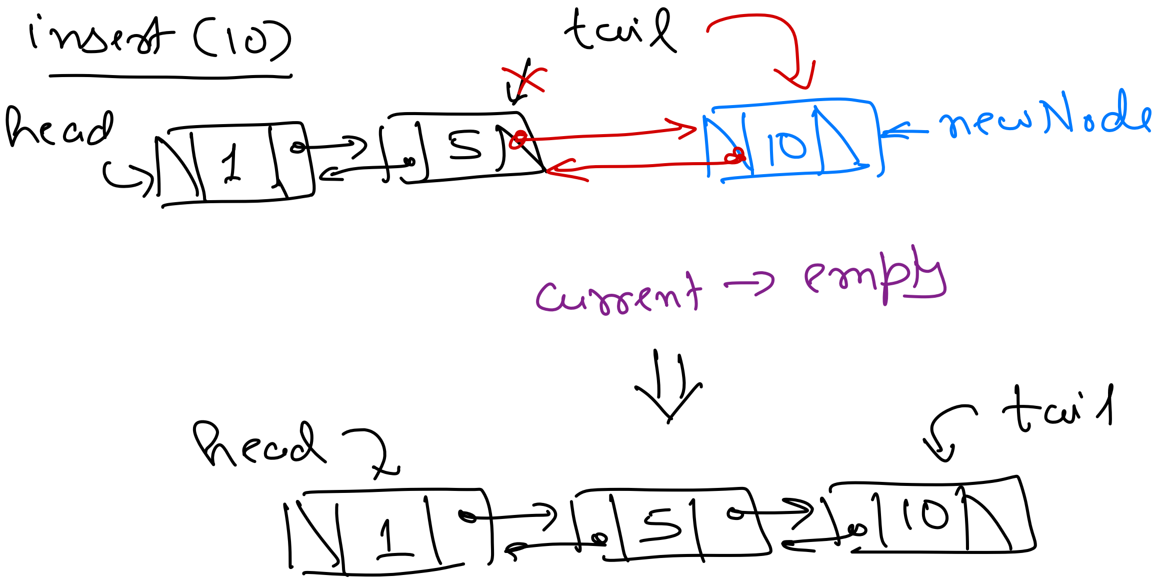
```
Insert( element )
// 1. Create new node
- Make memory for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next and previous to empty.

// 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.
```



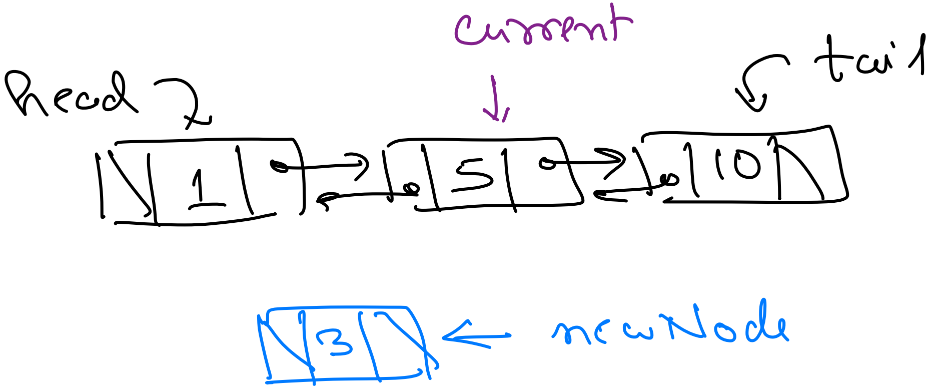


```
// 4. If adding before the first node? - Current is the first node.
- if (current is head) then
  - Before the first node comes newNode. // Set head's previous to newNode.
  - After newNode comes the first node. // Set newNode'next to head.
  - Make newNode as the first node. // Set head to newNode.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

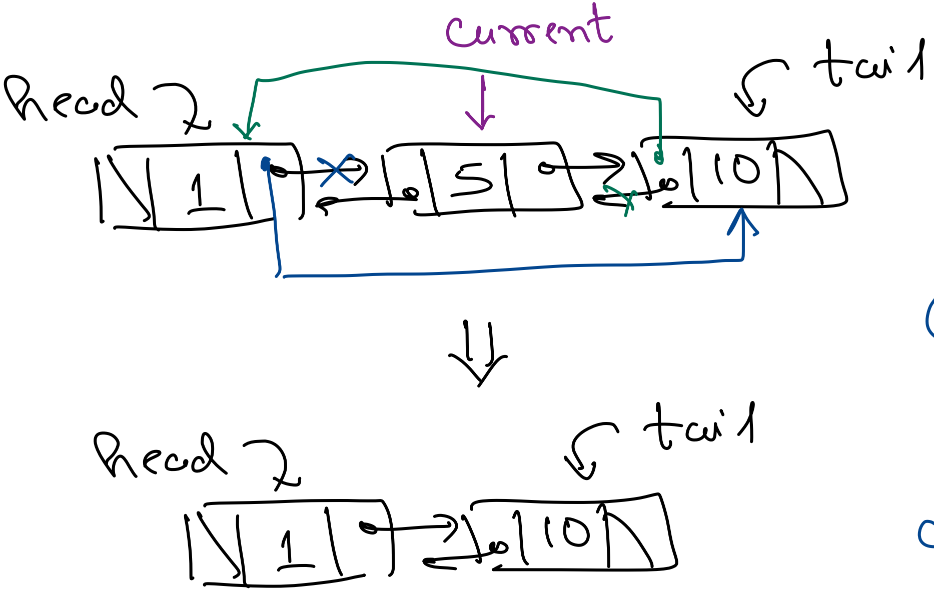
insert (10)



insert (3)



Delete Node from Doubly List



delete (5)

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

current . previous . next = current . next;

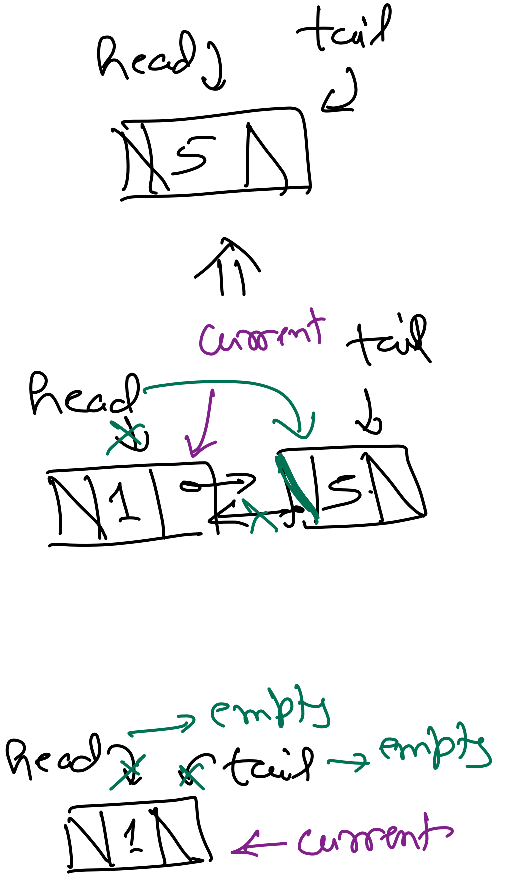
② Set current node's next node's previous to current's previous.

current . next . previous = current . previous

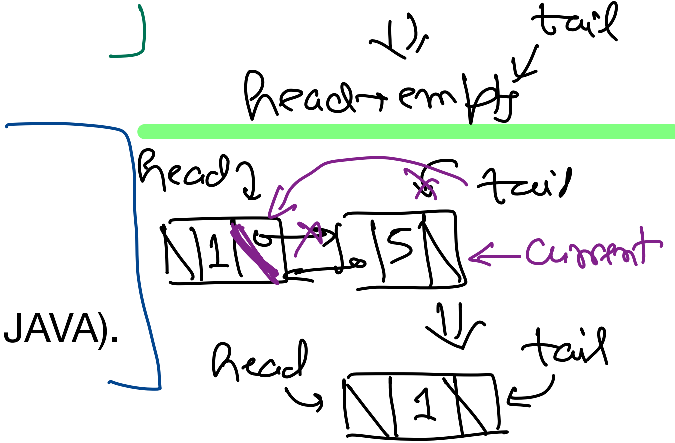
Special Cases

1. Empty list.
2. Deleti first element.
3. Deleti last element.
4. Delete element from list having only one node.
5. Element not found.

```
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?   #1
- if (current is empty) then
  - Stop.                                   #5

// Delete first node?   #2
- if (current is first node) then
  - Move head to head's next node.
  // Has the list become empty => list has only 1 node  #4
  - 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?     #3
- 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.
```

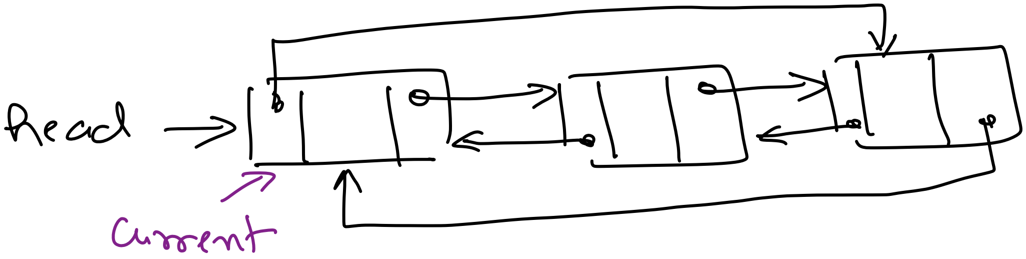


- Make current's next node come after current's previous node. // **Set current node's previous node's next to current node's next node.**
- Make the current node's previous node come before the current node's next node. // **Set current's next node's previous to current's previous node.**
- Release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

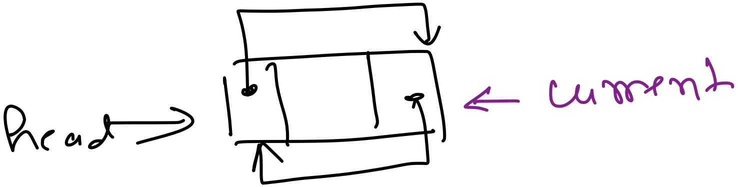
Circular Doubly Linked List

How to implement it?

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

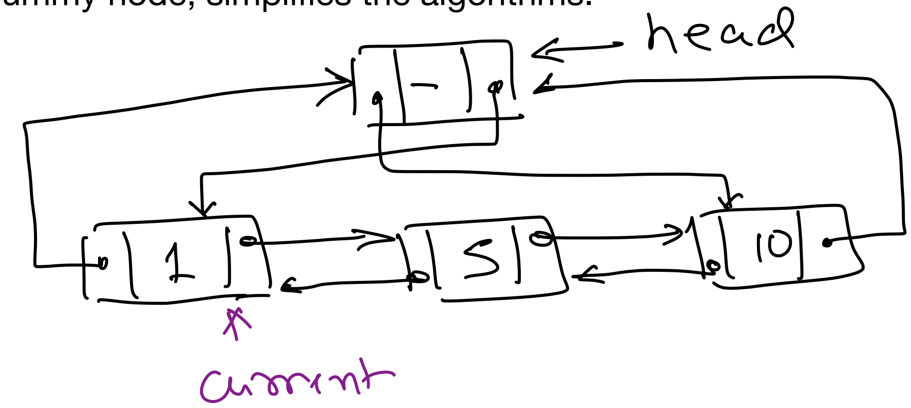


head → empty ← current

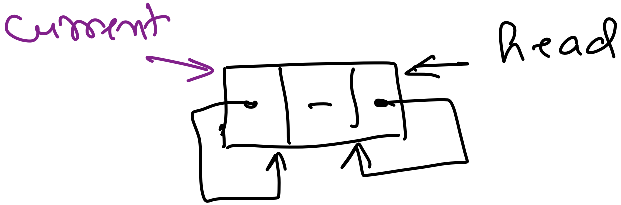


```
Traversal

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

How using a dummy node, simplifies the algorithms.



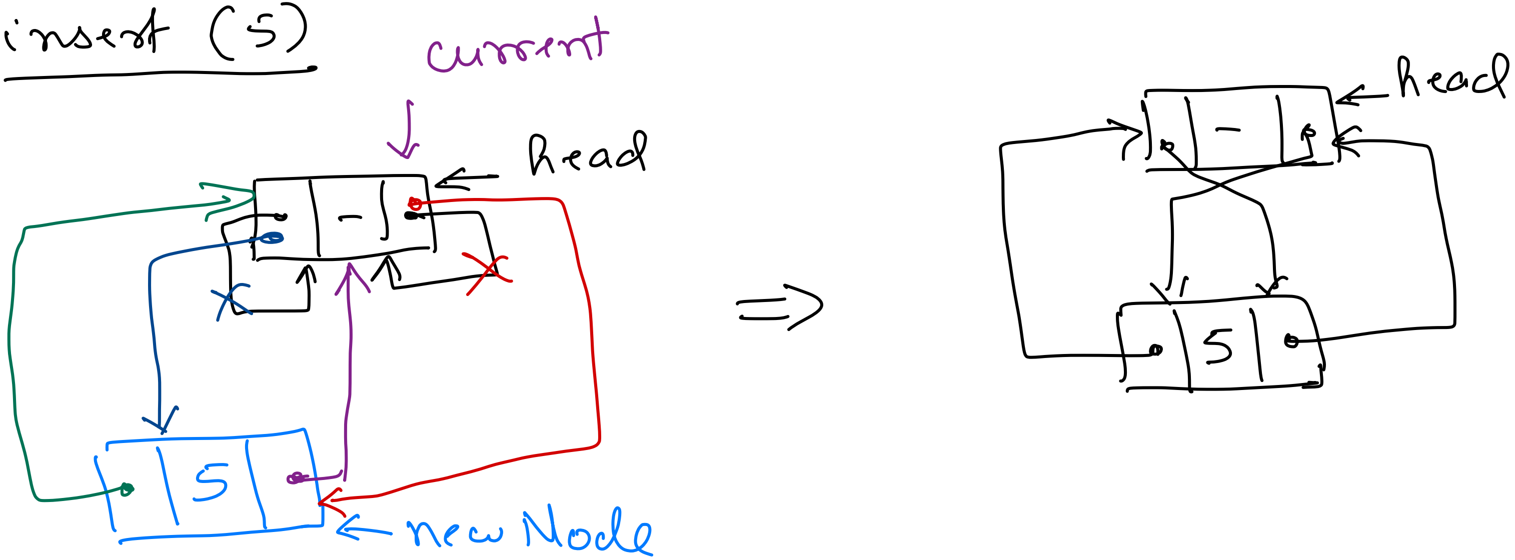
Empty circular list



Traversal (fwd)

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

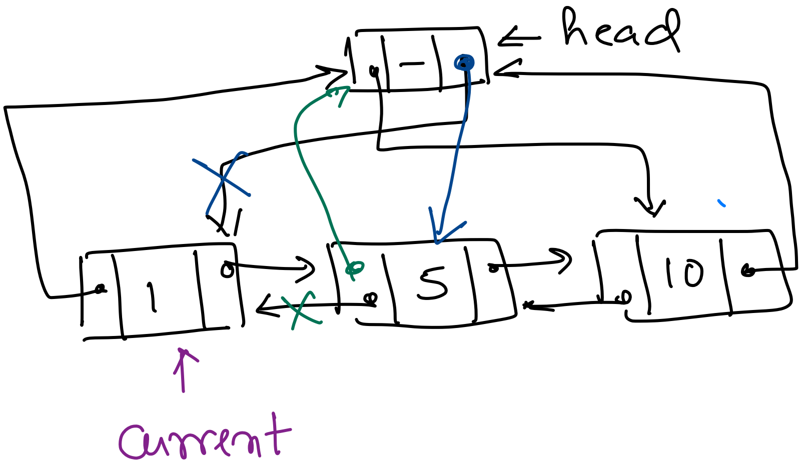
insert (5)



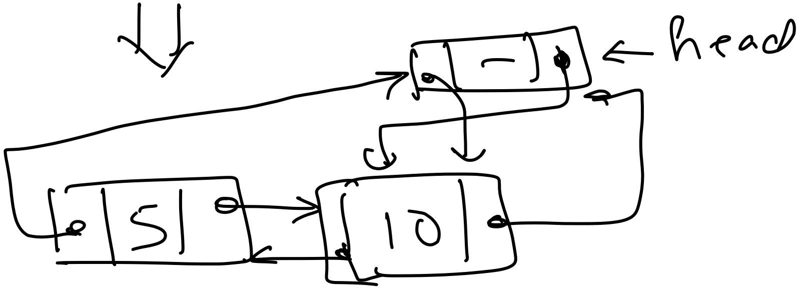
```
newNode.next = current,
newNode. previous = current. previous
Current. previous. next = newNode.
Current. previous = newNode;
```

```
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.

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

// 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(1)



```
current . previous . next = current . next;
Current . next . previous = current . previous;
```

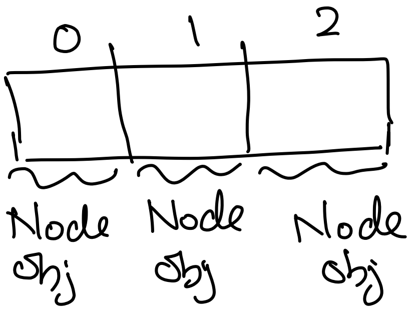


```
Delete (element)
// Find the node to be deleted - current node
- Set current to f~~irst node (head)~~ head.next
- while (current is not ~~empty~~) do head
  - if (current node’s data = element) then
     // Found the node - end the traversal.
     - End the traversal.
  - Move current to current’s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

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

How to use an array to allocate memory for all nodes for a linked list?

Array of objects in C++

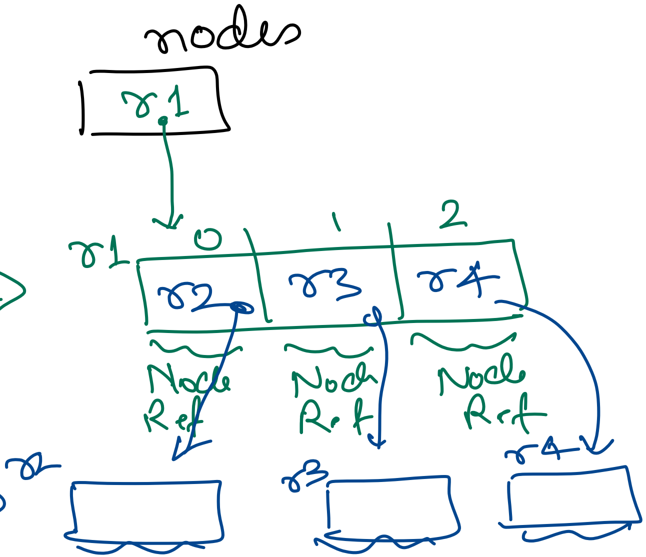
```
Node nodes[3]; ⟹
```



Array of objects in JAVA

```
Node [] nodes; ⟹
```

```
nodes = new Node[3]; ⟹

for (i=0; i<3; ++i)
    nodes[i] = new Node(); ⟹
```



Node objects

Object Pool

```
class NodePoolMgr {
    Node [] nodes;
    boolean [] isFree;
    Node createNode();
    void deleteNode(Node node);
}
```

```
AddAtFront(element) - Optimised
- Make space for new elements, say newNode. → Node newNode = nodePoolMgr.createNode();
- Store element in newNode’s data.
- Set newNode’s next to head.
- Set head to newNode.
- if tail is empty then
    - Set tail to head.
- Stop.
```

Recursion

When the solution of a problem is defined as a solution of a subproblem.

In programming - When a function calls itself.

Base case

$$n! = \begin{cases} \boxed{1, \text{ if } n = 0 \text{ or } n = 1} \\ n \times (n-1)!, \text{ otherwise} \end{cases}$$

Terminating condition

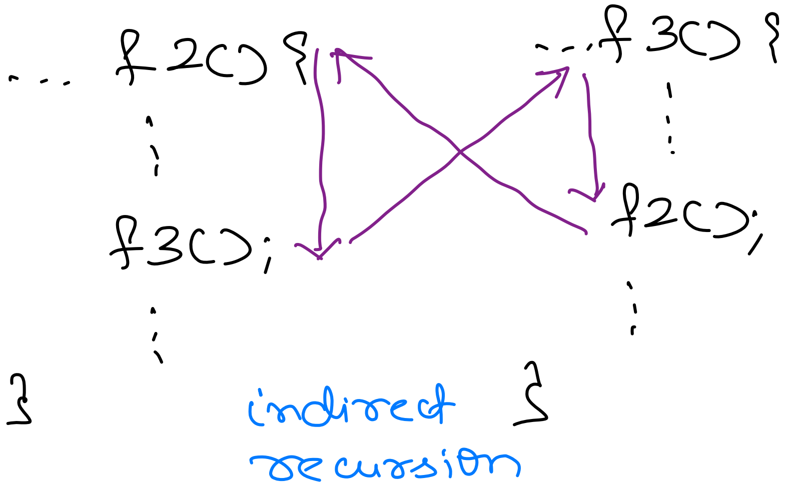
```
int factorial (int n) {
    if ((n==0) || (n==1)) {
        return 1;

    return n * factorial (n-1);
}
```

Direct vs Indirect recursion.

Infinite recursion and terminating condition/base case.

```
... f1() {
      :
    f1();
     :
}
```

← direct recursion



```
... f2() {
     :
    f3();
     :
}
```

```
... f3() {
      :
    f2();
     :
}
```

indirect recursion

```
... f4() {
      f4();
   }
```

← infinite recursion ⇒ when recursive call is made before terminating condition.