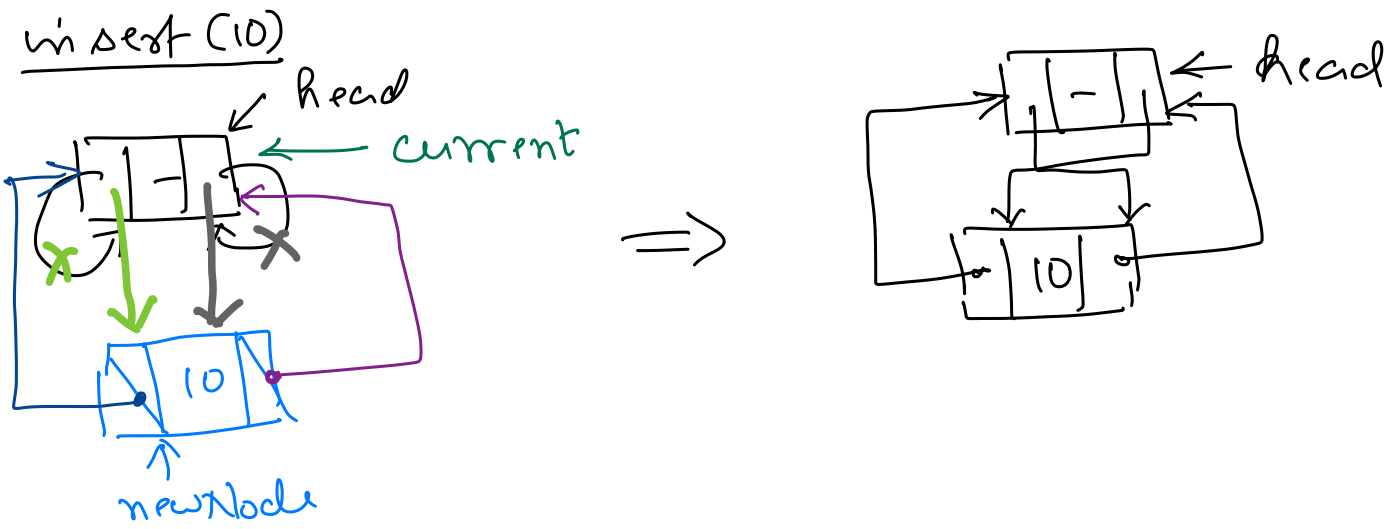
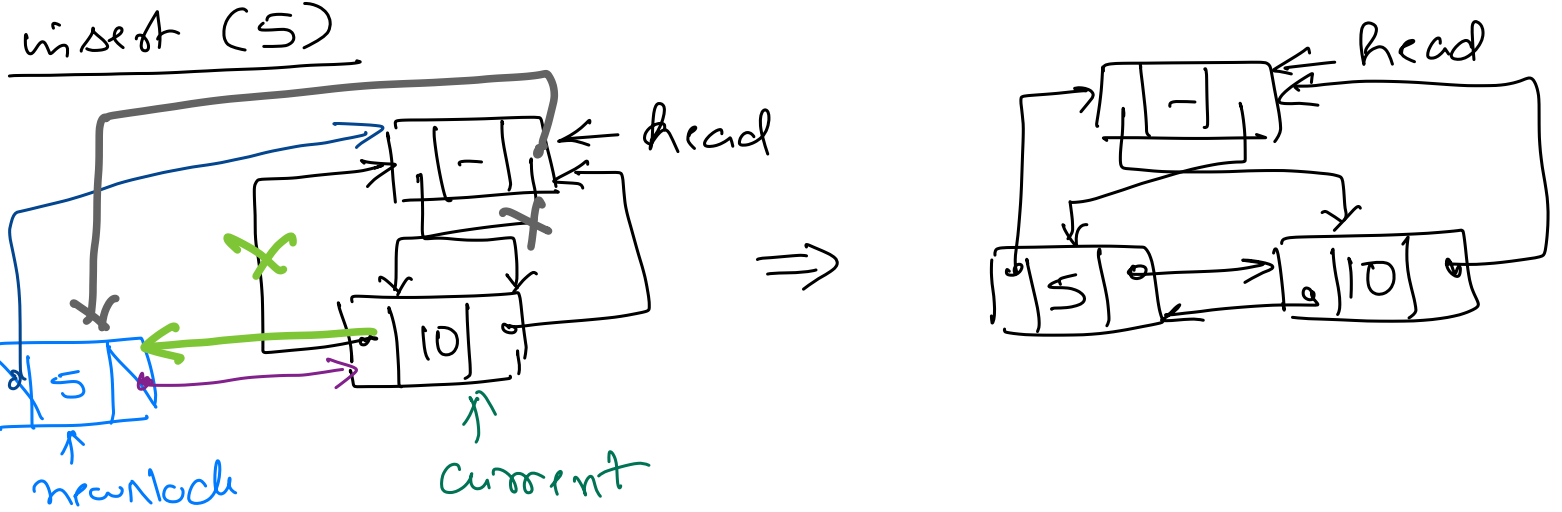
Insert(element)

// 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.

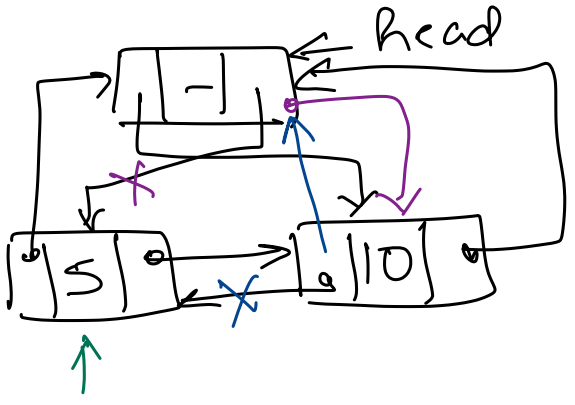
// 3. Traverse list to find node - current node.
- Set current to ~~head~~ (first node). head's next
- while (~~current is not empty~~) do current is not head
  - if (current node's data > element) then
    // Found the node, end the traversal.
    - End the traversal.
  - Set current to current's next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

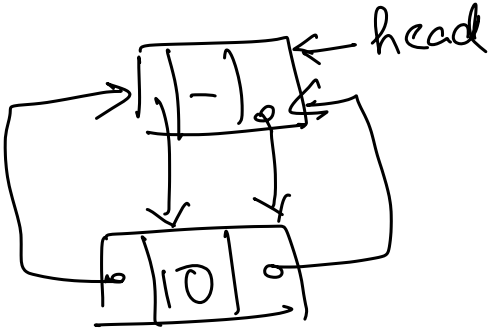
// 6. Add a new node between current and current's previous node.
- Make the current node come after newNode. // **Set newNode's next to current.**
- Make the current node's previous node come before newNode. // **Set newNode's previous to current node's previous.**
- Make newNode come after the current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


insert (10)
head
current
-
10
newNode
head
-
10

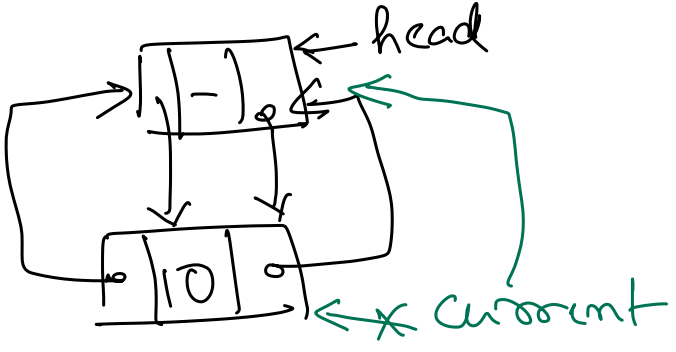

insert (5)
head
-
10
5
newNode
current
head
-
5
10

delete (5)





delete (20)



Delete (element)
// Find the node to be deleted - current node
- Set current to first node (~~head~~) head's 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.

Iterator → mechanism using which we can access elements stored in a data structure, one by one.

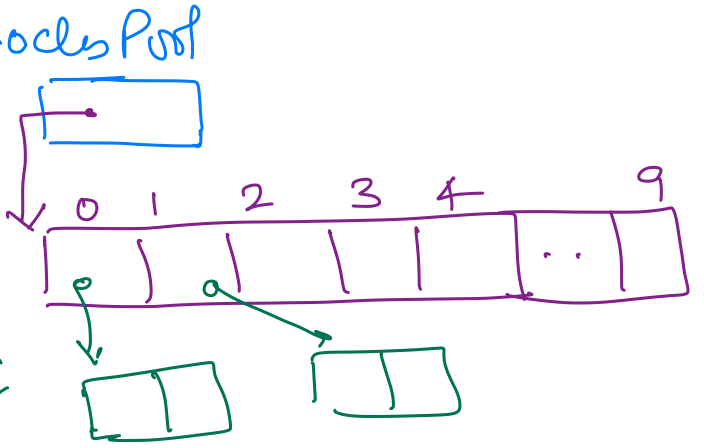
Iterator<T> interface in JAVA.

Iterable<T> interface in JAVA

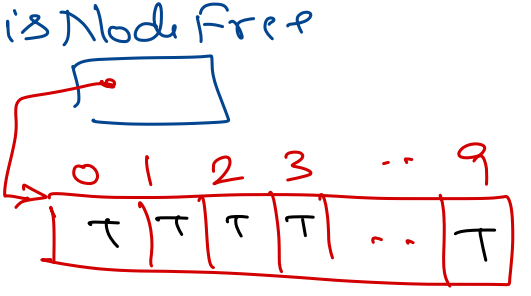
Class implementing these interfaces, their object can be used in for each loop.

How to use an array to allocate memory for all nodes for a linked list? ← Object Pooling.

```
Node [] nodesPool;

nodesPool = new Node[10];
for (int i=0; i<10; ++i) {
    nodesPool[i] = new Node();
```




```
boolean [] isNodeFree;

isNodeFree = new boolean[10];
for (int i=0; i<10; ++i)
    isNodeFree[i] = true;
```



class NodePoolManager {

Node createNode();

void deleteNode(Node);

}

NodePoolManager nodePool;

AddAtFront(element) - Optimised
- Make space for new elements, say newNode. → Node newNode = nodePool.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.

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

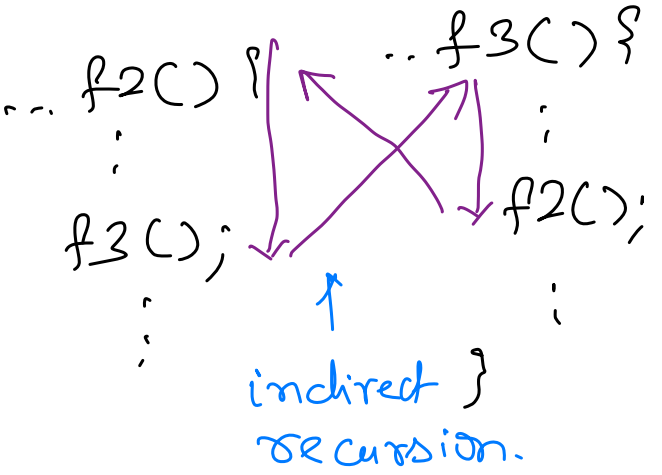
```
int factorial ( int n) {
    if((n==0) || (n==1))
        return 1;

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

terminating condition.

Direct vs Indirect recursion.

Infinite recursion and terminating condition/base case.

```
... f1( ) {
      :
    f1();   <- direct recursion
      :
}
```




```
... f4() {
      f4();   <- Infinite recursion. =>
      :
}
```

when recursive call is made before terminating condition.

Divide and Conquer

If a problem can be divided into smaller problems such that, the solution of smaller problems gives us the solution of bigger problem.

DIVIDE - Divide larger problem into smaller problem.
CONQUER - Solve each smaller problem until they are a base case.
COMBINE (Optional) - Combine solution of smaller problems to find solution of larger problem.

Backtracking

Build solution step by step.
At each step, we discard steps(s) that do not result in a solution.

→ Solve maze.

→ Sudoku

→ 8 queen problem.

→ knight tour.

Multiply two numbers, without using * operator.

$$multiply(m, n) = \begin{cases} 0, & \text{if } m = 0 \text{ or } n = 0 \\ m, & \text{if } n = 1 \\ n, & \text{if } m = 1 \\ m + multiply(m, n-1), & \text{otherwise} \end{cases}$$

$m \times n$

$\Rightarrow \underbrace{m + m + m + \ldots + m}_{n \text{ times}} \Rightarrow m + \underbrace{(m + m + \ldots + m)}_{(n-1) \text{ times}}$

$\Rightarrow \underbrace{n + n + n + \ldots + n}_{m \text{ times}}$

$m \times (n-1)$

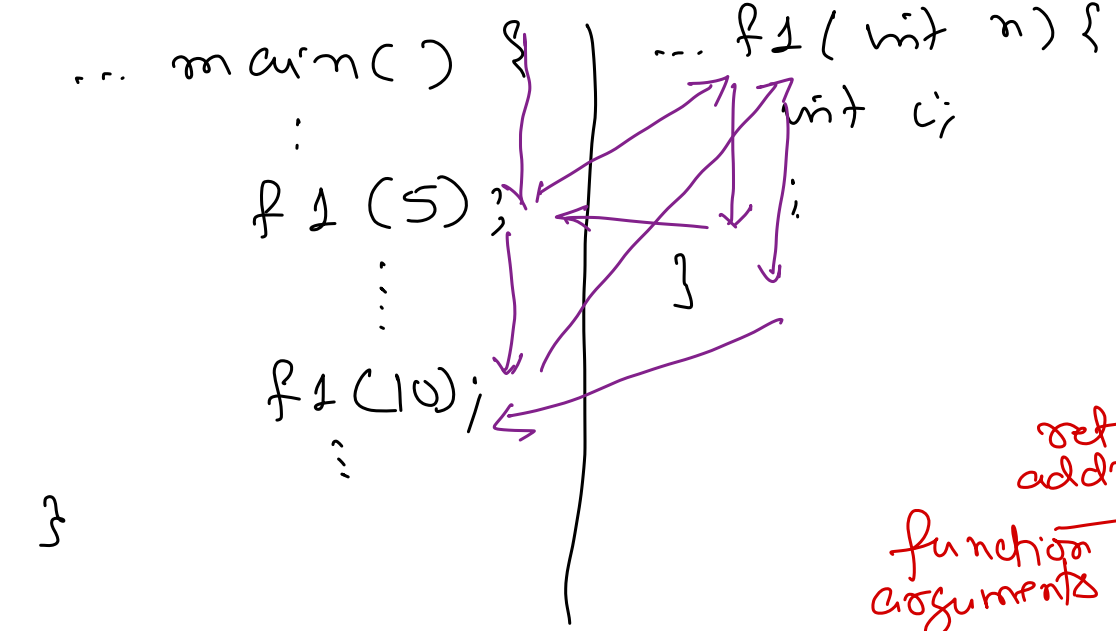
Assignment: Define multiplication for negative numbers.

$m = 5$
$n = 102197$ ← Handle such scenario efficiently in algorithm.

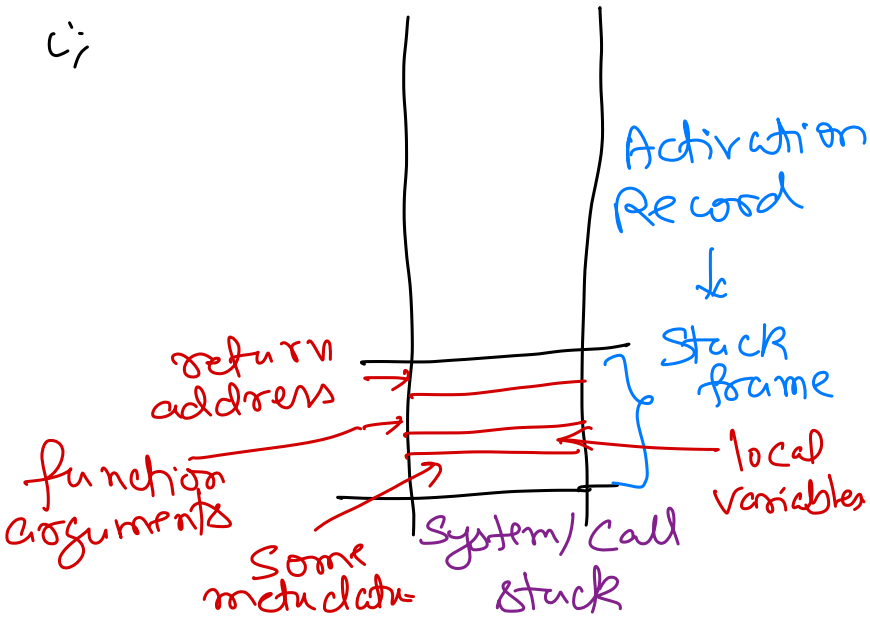
Assignment: Divide two numbers
→ give quotient ⇒ QUOTIENT(m, n)
→ give remainder ⇒ REMAINDER(m, n)

How do function call works

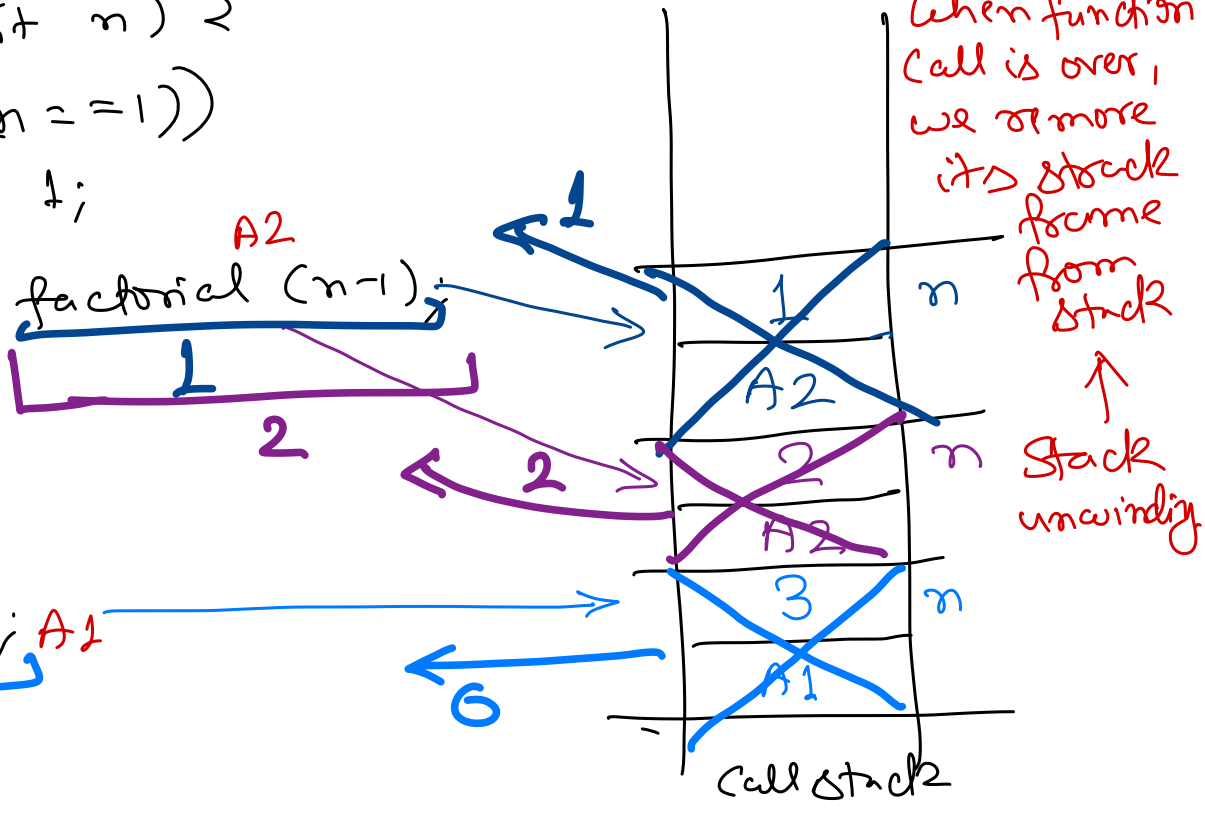
```
... main() {
    :
    f1(5);
    :
    f1(10);
    :
}

... f1(int n) {
    int i;
    :
}
```



Stack (System / Call Stack)



```
int factorial ( int n) {
    if((n==0)||(n==1))
        return 1;
    return n * factorial (n-1);
}
... main() {
    :
    factorial(3);
    :
}
```



To return value for function general purpose CPU registers are used.

Calling convention → to invoke function.

!!

→ in which order function arguments are pushed on stack

→ who cleans up stack frame after function call is over.