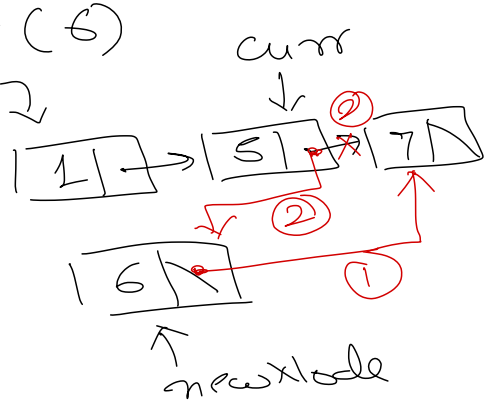
Insert in singly linked list using one pointer.

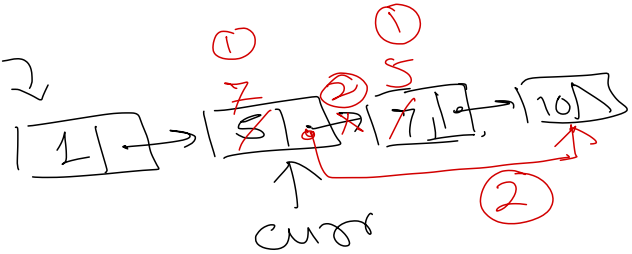
Insert (6)



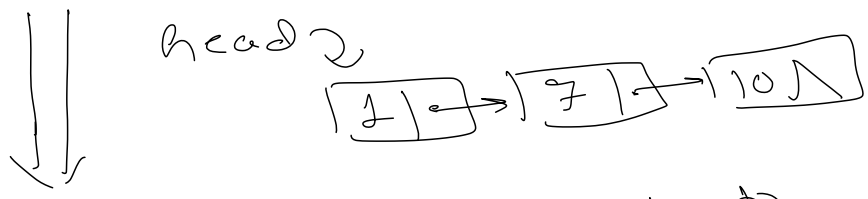
check if element is less than current's next node's data

① Set newNode's next to curr's next.
② After current node comes newNode.

Delete node from singly linked list using one pointer.



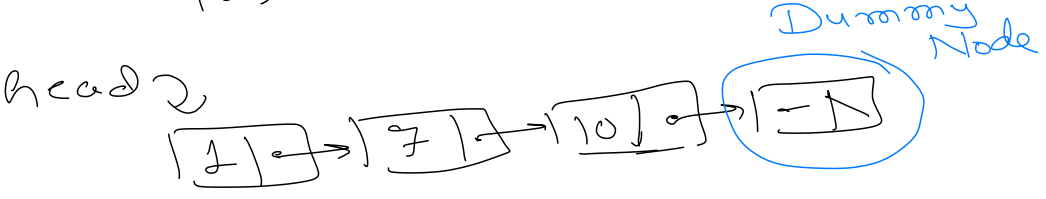
① Swap data of current & current's next node.
② Delete current's next node.

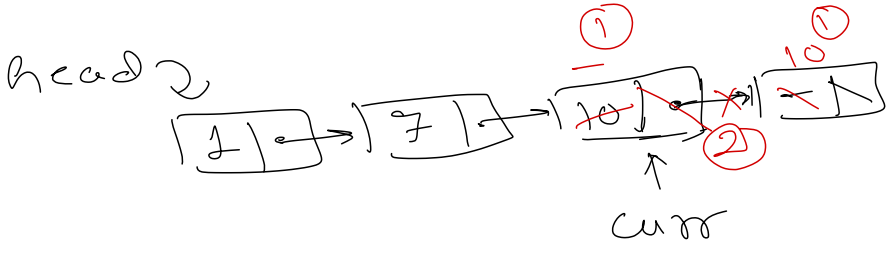


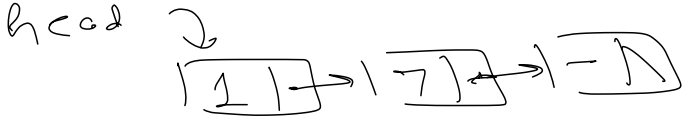
Set current node's next to (current node's next) node's next

Limitation: Can not delete last node.

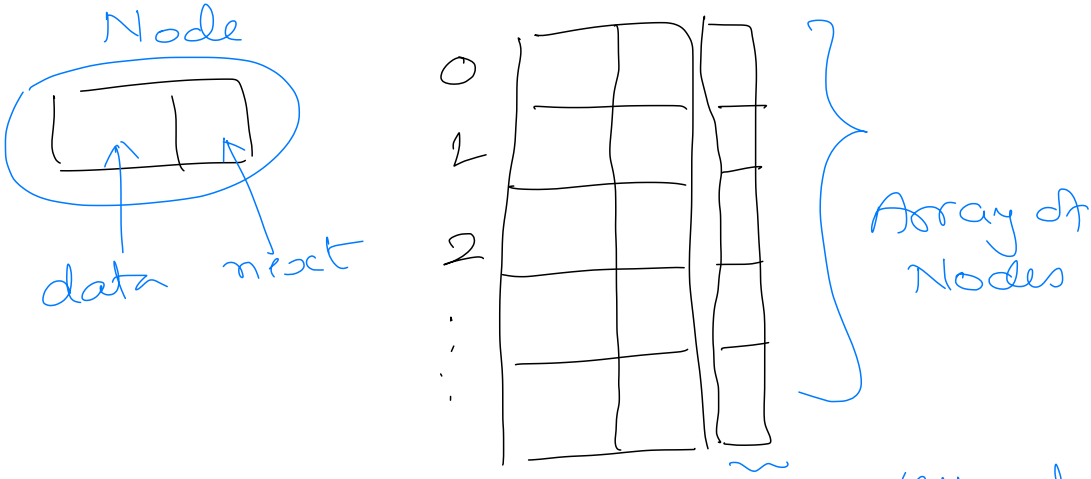
Solution? Maintain a dummy node after last node.



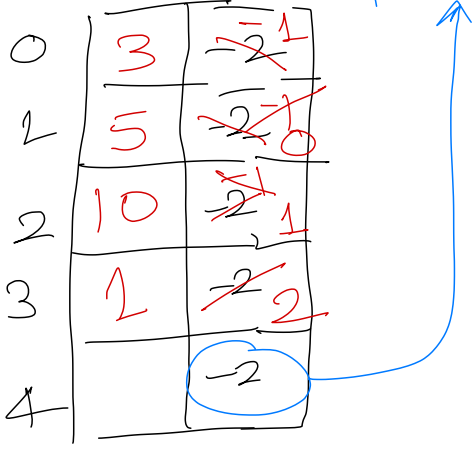




Implement Linked List Using Arrays



Indicates node is free ⇒ Not part of a list



(2) ~~Add~~ AtFront(5)

head = ~~0~~ 1

newNode = 1

(3) ~~Add~~ AtFront (10)

head = ~~1~~ 2

newNode = 2

Size = 5 head = (-1) ⇒ empty list

(1) AddAtFront(3)

newNode = 0

head = ~~-1~~ 0

AddAtFront(element) - List having only head pointer.
- Make space for new element, say newNode.
- Store element in newNode's data.
- Set newNode's next to empty.
- if list is empty then
  - Set head to newNode.
  - Stop.
- Set newNode's next to head.
- Set head to newNode.

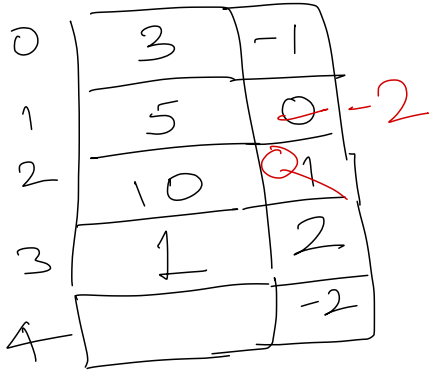
(A) AddAtFront (2)
head = 2 3
newNode = 3

Serialization is easy

Serialize: Linked list using dynamic memory - We traverse list and store element, one at a time, to file.

De-serialize: Read element, one at a time from file, and add to list.

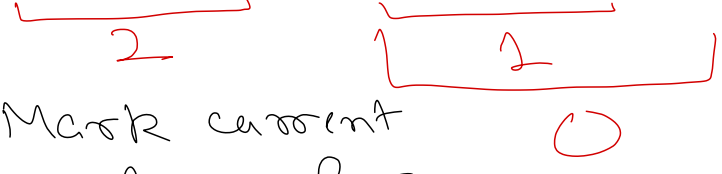
Serialize - List using array - We write entire memory block in one go.
Des-serialize: Read contents of entire array.



(1) Delete (5)
head = 3

curr → 3 2 1
prev → 1 3 2

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.



- Mark current node as free.

RECURSION

Recursion - When a function call itself.

```
F1C){
   F2C);
}
```

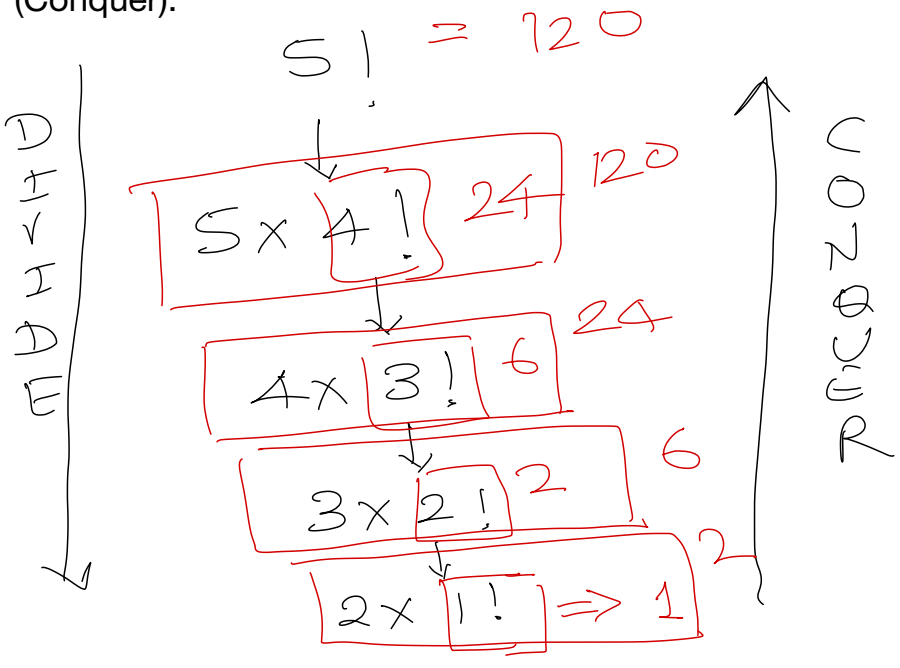
```
F1C){
   F1C);
}
```

Recursion - when the solution of a problem is defined as a solution of sub-problem.

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

Terminating Condition

Divide and Conquer - Break the bigger problem into two or more smaller problems and solve the smaller problems (Divide). We combine the solution of smaller problem(s) to define the solution of larger problem (Conquer).



→ Product of two numbers without using multiplication.

int Mul(int x, int y); x & y are both numbers

$$Mul(x,y) = \begin{cases} 0, & \text{if } x=0 \text{ OR } y=0 \\ x, & \text{if } y=1 \\ y, & \text{if } x=1 \\ x + Mul(x, y-1) \end{cases}$$

→ Divide two whole numbers without using division.

```
void F1() {
    System.out.println(...);
    F1(); ← Infinite recursion
}
void F2(int i)
    if (i==0)
        return; } → Terminating condition
    System.out.println(...);
    F2(i-1); ← Recursive call.
}
```

F2(-1); ⇒ result in infinite recursion

→ Direct Recursion

↳ When a function calls itself directly.

```
void F2(){
    :
    F2();
}
```

→ Indirect Recursion

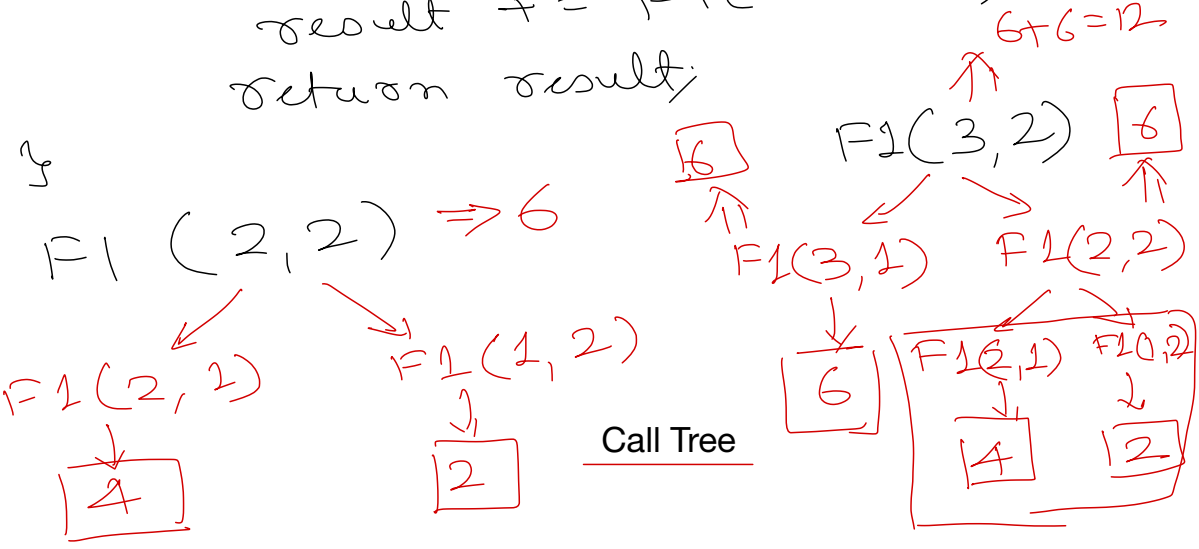
```
void F1(){
    :
    F2();
}

void F2(){
    :
    F3();
}

void F3(){
    :
    F1();
}
```

```
int F1( int x, int y) {
if (x == 1)
    return y;
if( y == 1) return 2*x;

int result;
result = F1( x, y-1);
result += F1( x-1, y);
return result;
}
```

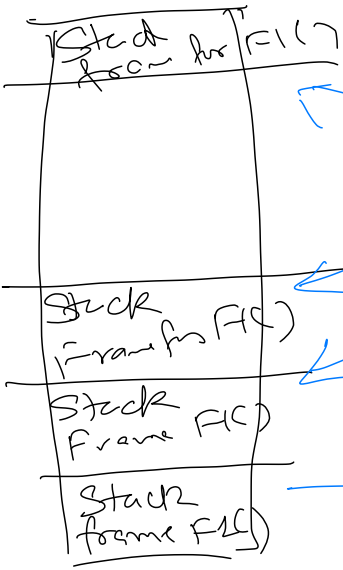


Call Tree

→ Call Stack ⇒ Stack of stack frames.

Consists of
→ Function arguments
→ Local variables
→ Return address

Created for every function call.



Call Stack

```
void F1(){
   :
   F1();
}
```

F1();

Terminates due to Stack Overflow as Call Stack is having no memory left for new stack frames.

```
int F1( int x, int y) {
  if (x == 1)
      return y;
  if( y==1) return 2*x;

  int result;
  result = F1(x, y-1);   (2)
  result += F1(x-1, y);  (3)
  return result;
}
```

```
F1(2,2); (1)
   .
   .
   .
F1( ... ); (4)
   .
   .
   .
```

(6)

top

x	result
1	x
2	y
(3)	R.A.
x	result
2	x
1	y
(2)	R.A.
x, 4, 6	result
2	x
2	y
(1)	R.A

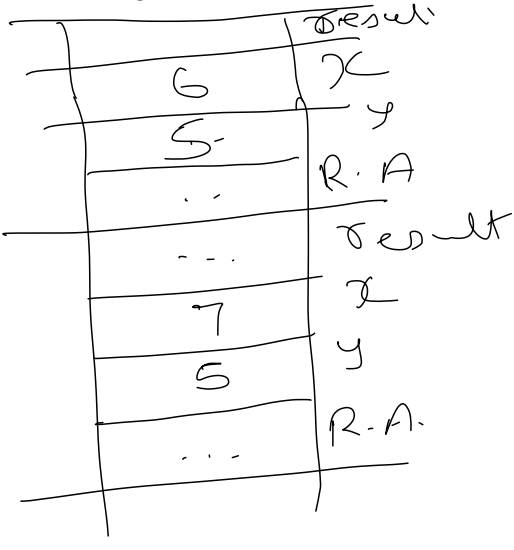
Tail Recursion –

Recursive call that's the last statement in function.

```
int F1( int x, int y) {
  if (x == 1)
      return y;
  if( y == 1) return 2*x;

  int result;
  result = F1(x, y-1);
  result += F1(x-1, y);
  return result;
}
```

tail Recursion



Tail Recursion optimization.

void f1()

⋮

f1();

}

f1();