Types :

Single linked list
Doubly linked list
Circular linked list

- Single

two parts $\left.\begin{array}{l}\text{Data}\\ \text{Link}\end{array}\right\}$ nodes

Self referential structure : contains a pointer to a structure of the same type.

```
struct abc
   int a
   struct abc *self
```

Creating :

1) stdio.h = Standard input output
2) stdlib.h = for malloc
3) Declarating the head :

```
struct node *head = NULL;
```

head = $\underbrace{\texttt{(struct node*)}}_{\text{Not mandatory}}$malloc(sizeof(struct node));

the node's pointer will be store in head var.

The one line form is :

```
struct node head* = malloc(sizeof(struct node))
```

4) Giving data and next, exist two ways

`(*head).data = 34;` or `head->data = 34;`

5) Creating another is very similar but updating the latest node

```
struct node *current = malloc(sizeof(struct node));
current->data = 95;
current->next = NULL;
head->next = current;
```

↓

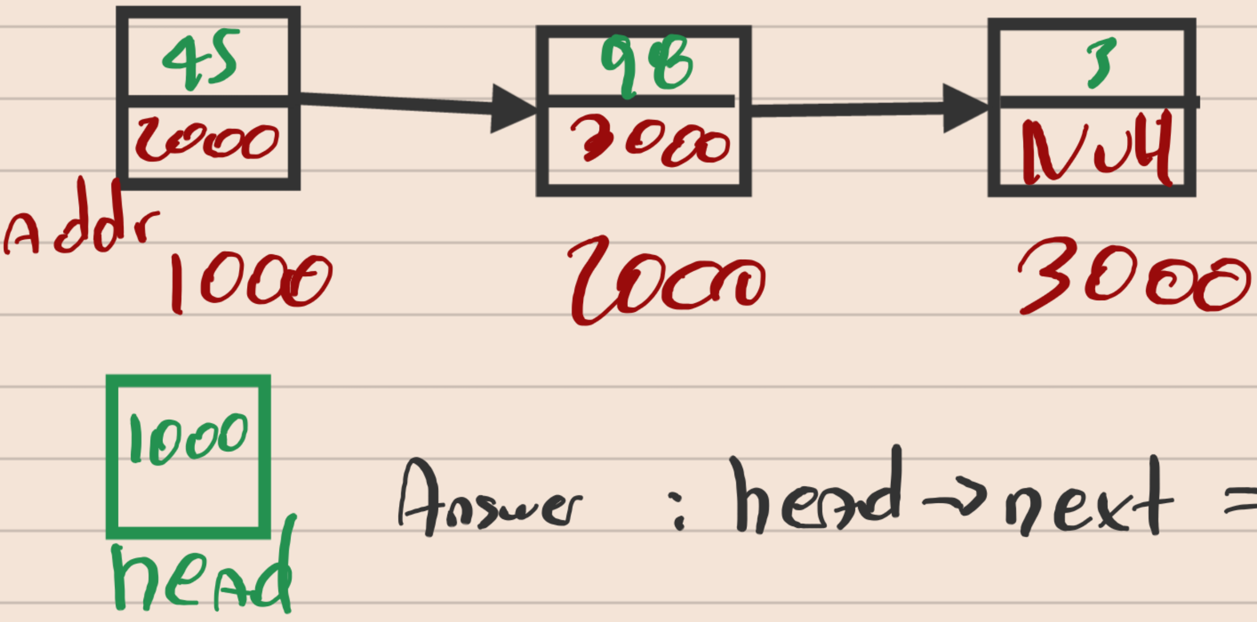
Recall it's pointing to NULL, now is pointing to the new node.

* tip : use if check in every malloc func.

```
if (current == NULL)
    return 1;
```

Method 2 :

1) what does head->next give?



Answer : head->next = 2000

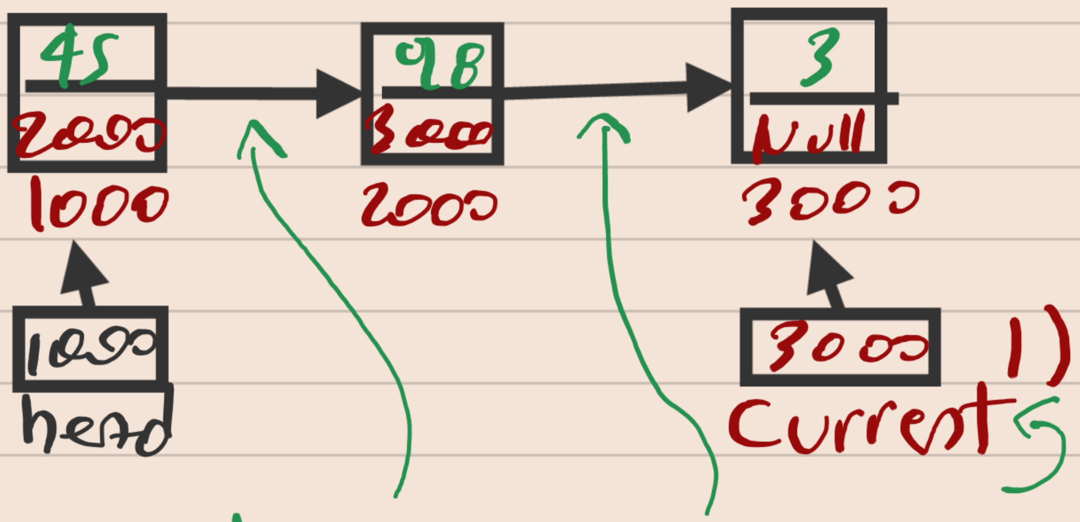
2) and head->link->link?

Answer : gives = 3000

head->link->link->link = NULL

Key word of method 2 is reuse,

head->link->link = current



head->next->next = current

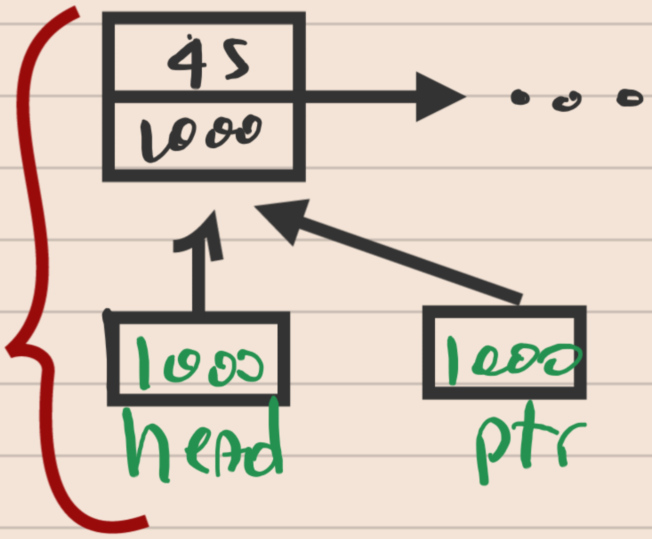
equal to 3000

Traversing :

visiting each nodes

In this case, counting nodes, we creates a func. geting the beginning of the node:

```
        countNode(head);
                  Beginning.
int count = 0          // counter
if (head = NULL)
   printf("empty)
Struct node ptr = head
while (ptr != null)
      count ++
      ptr = ptr->link
```



Printing every data:

Using the traversing technique:

```
while (ptr != NULL)                    ②
      printf("%d", ptr->data)
      ptr = ptr->next                  ①
```

① ptr is overwrite every looping time until it arrives to a Null ptr->next.
② stops when the tail is found.