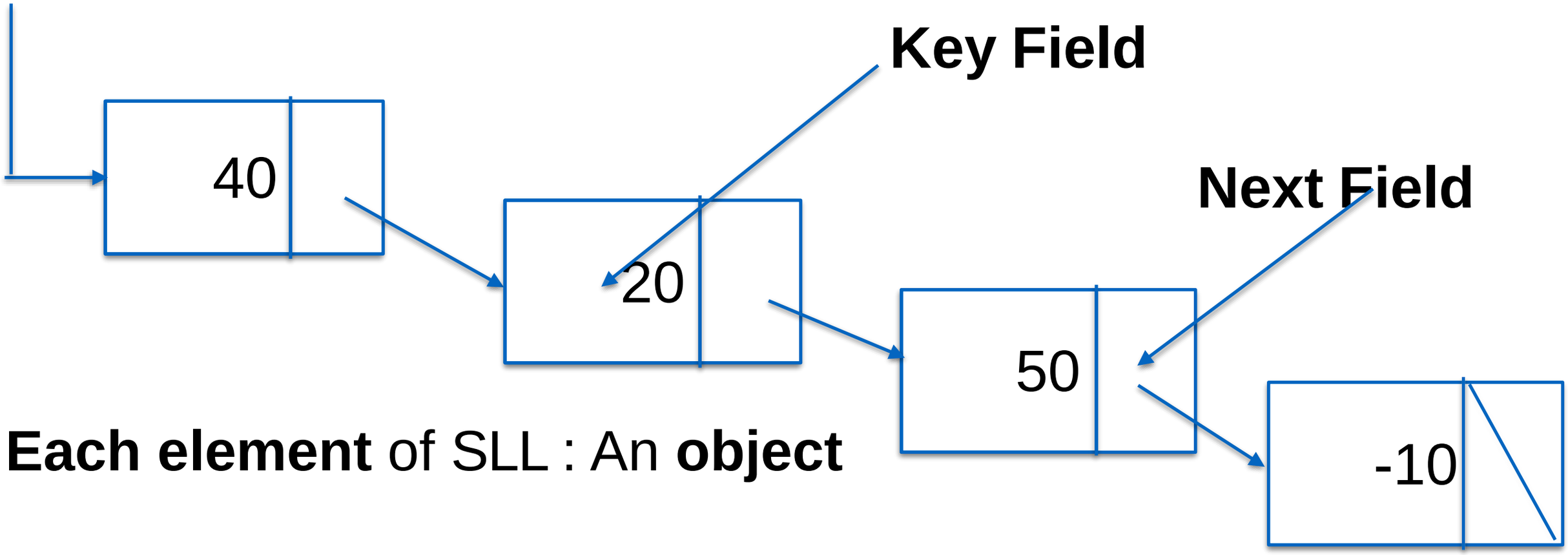
Linked Lists

Linked Lists

- **Linked list** is a data structure in which objects are arranged in a linear order
  - Linear order is determined by a pointer in each object
- Provides a **simple, flexible representation for dynamic sets** and it supports all the operations (query & modifications)
- **Different types of linked list**:
  - Singly Linked List (SLL)
  - Doubly Linked List (DLL)
  - Circular Linked List (CLL)

SINGLY LINKED LIST (SLL)

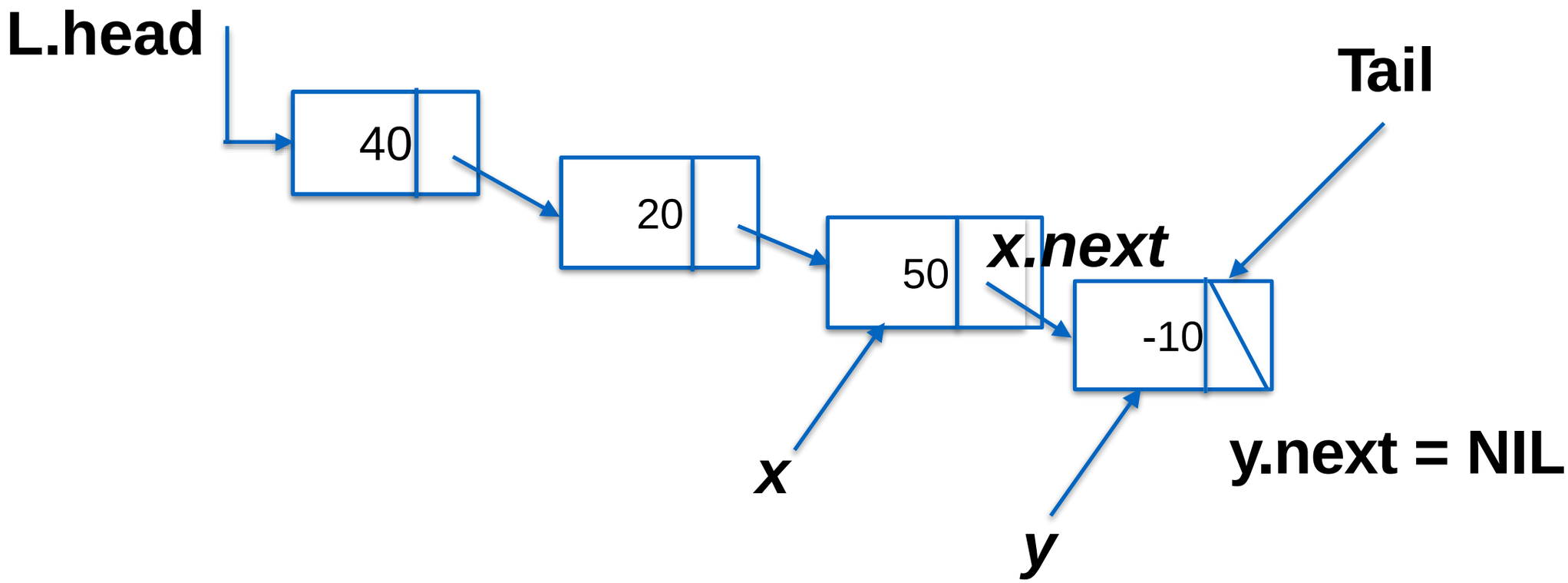


Each element of SLL : An **object**

Attributes: Key and a Next pointer

The overall linked list also visualized as an object in CLRS text book

SINGLY LINKED LIST (SLL)



- An attribute **L.head** points to the **first element** of the list.
  If **L.head = NIL, the list is empty** (in C code, we write L->head instead of L.head)
- Given an element *x* in the list, ***x.next*** points to its **successor** in the linked list (Remember in C code, we write x->next instead of x.next)
- If **x.next = NIL**, the element x has **no successor**

Dynamic Memory allocation functions - C Language

- Dynamic Memory Allocation functions - create amount of required memory.

- C language provides features to manage memory at run time

- 4 DYNAMIC MEMORY ALLOCATION FUNCTIONS IN C:
  - malloc()
  - calloc()
  - realloc()
  - free()

Dynamic Memory allocation functions - C Language

- Dynamic Memory Allocation functions - create amount of required memory.

- C language provides features to manage memory at run time

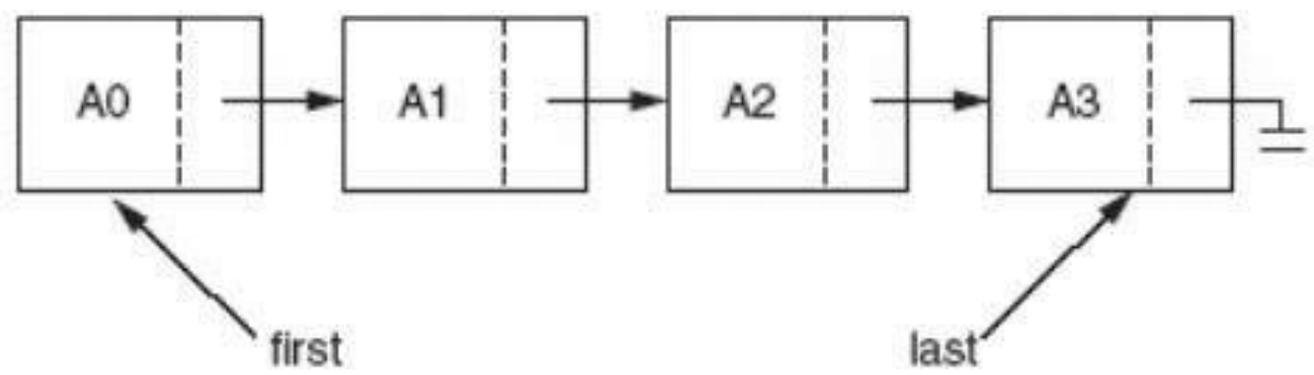
- 4 DYNAMIC MEMORY ALLOCATION FUNCTIONS IN C:
  - malloc()
  - calloc()
  - realloc()
  - free()

Linked list implementation details

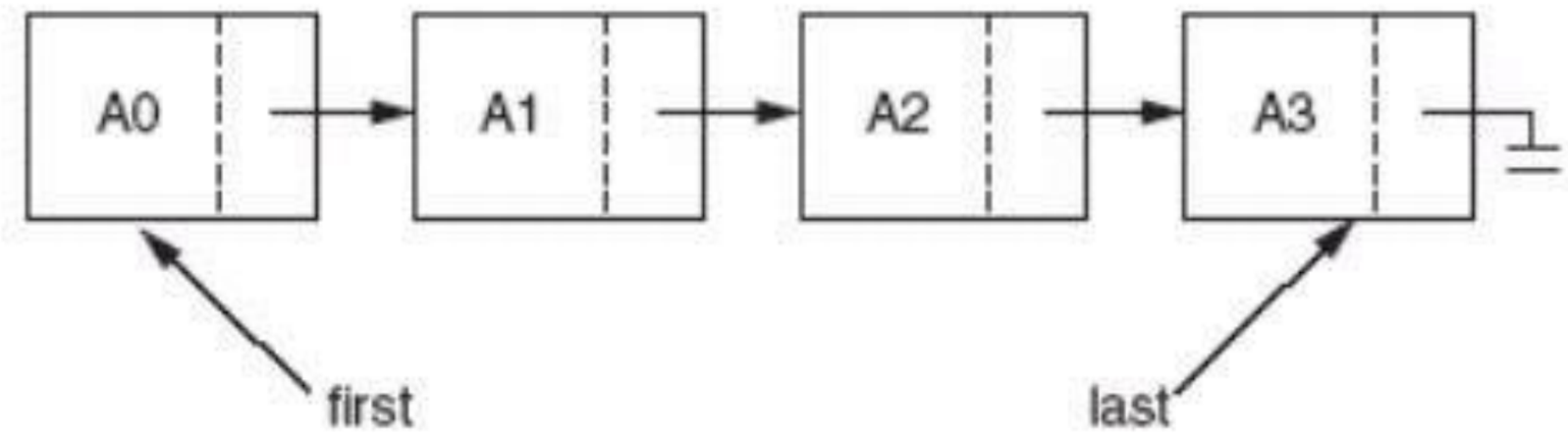
- A linked list is simply a chain of structures which contain a pointer to the next element and it is dynamic in nature.

- Items may be added to it or deleted from it.

- A list item has a pointer to the next element, or NIL if the current element is the tail (end of the list).



Example Linked List



- This pointer (pointer to the next element) points to a structure of the same type as itself.

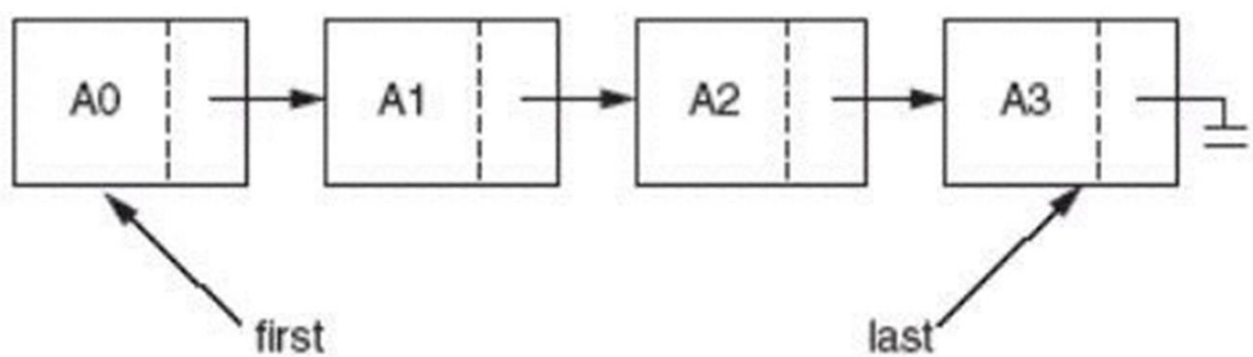
- This structure that contains elements and pointers to the next structure is called a Node.

- The first node is always used as a reference to traverse the list and is called HEAD. The last node points to NULL.

Declaring a node in Linked list

Declaring a node in a Linked list :

```
struct node
{
    long int key;
    struct node *next;
};
```



The above definition is used to create every node in the list.

The **key** field stores the key of the element and the **next** is a pointer to store the address of the next node.

Note that, in place of a data type, **struct node** is written before next.

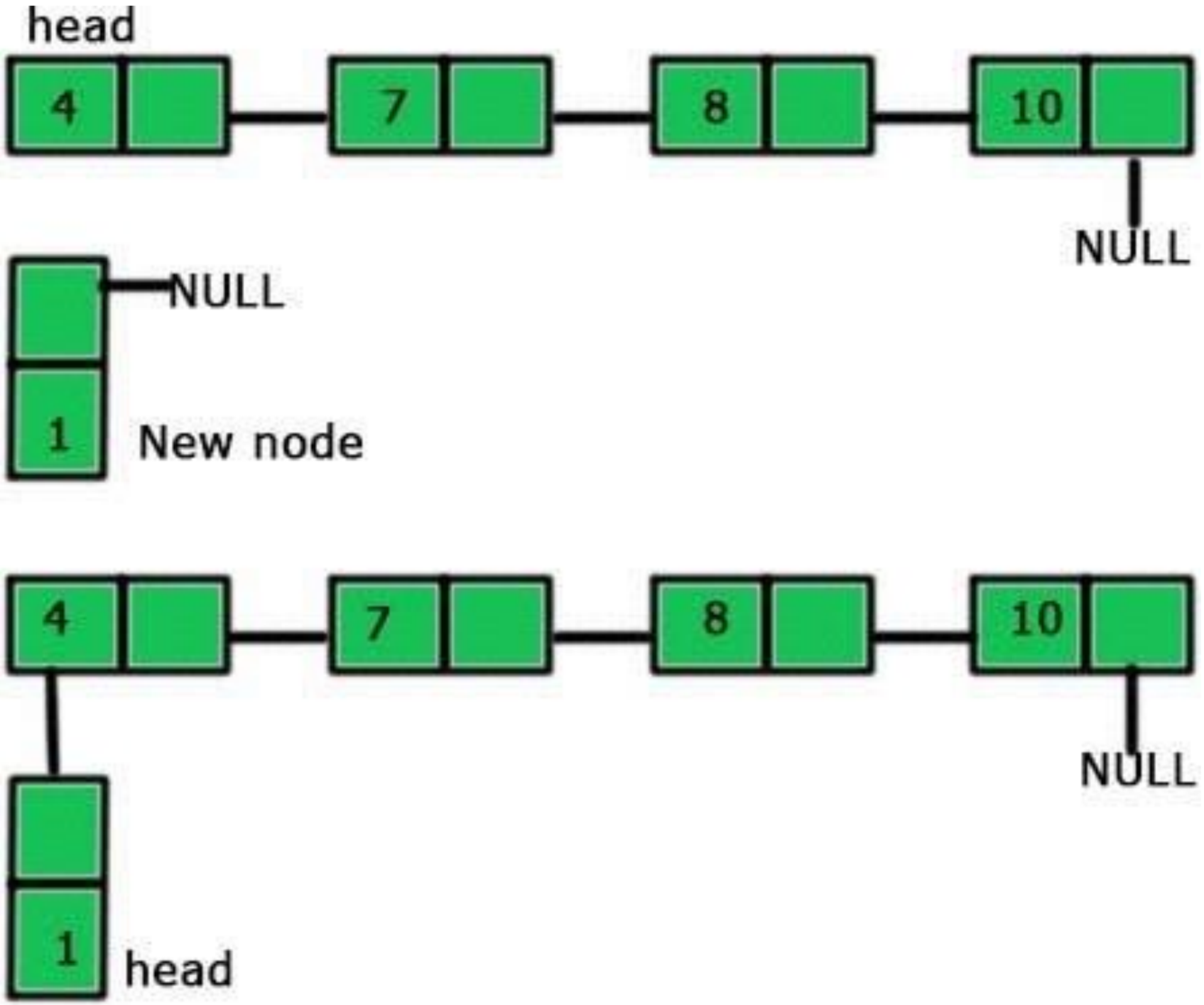
That's because its a **self-referencing pointer**. It means a pointer that points to whatever it is a part of.

Here, **next** is a part of a node in the linked list and it will point to the next node in the linked list

C Code for Creating a Node

```
struct node
{
    long int key;
    struct node *next;
};
typedef struct node *node; //Define node as pointer of data type struct node
struct LL //  LL stores a pointer to the head of the LL
{
        node head; // head is a pointer to the struct node
};
typedef struct LL *LL; //Define LL as pointer of data type struct LL
node CREATE_NODE(long int k)
{
        node temp; // temp is a pointer of type node
        temp = (node)malloc(sizeof(struct node)); // allocate memory using malloc()
        if(temp == NULL) //enough space is not allcated
                exit(0);
        temp->key = k;
        temp->next = NULL; // make next point to NULL
        return temp; //return the new node
}
```

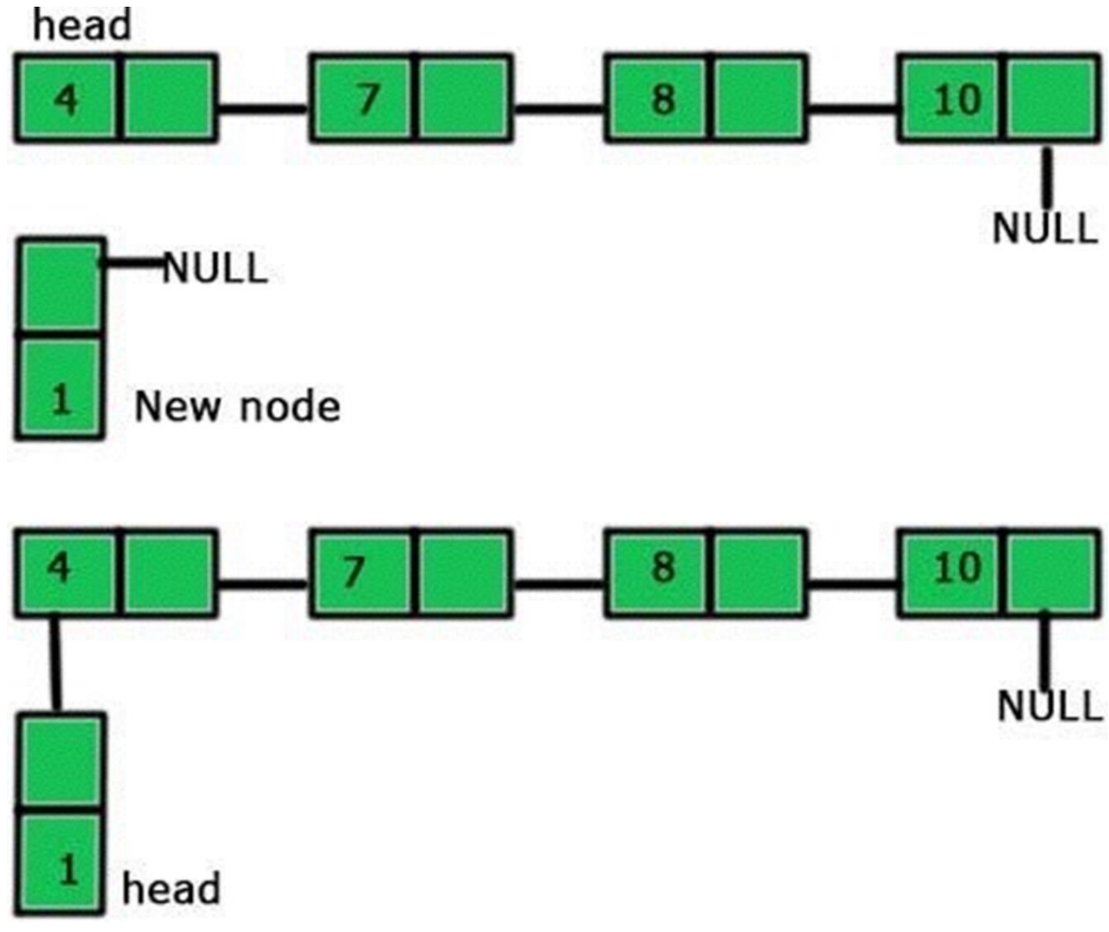
Insertion at the beginning of the linked list



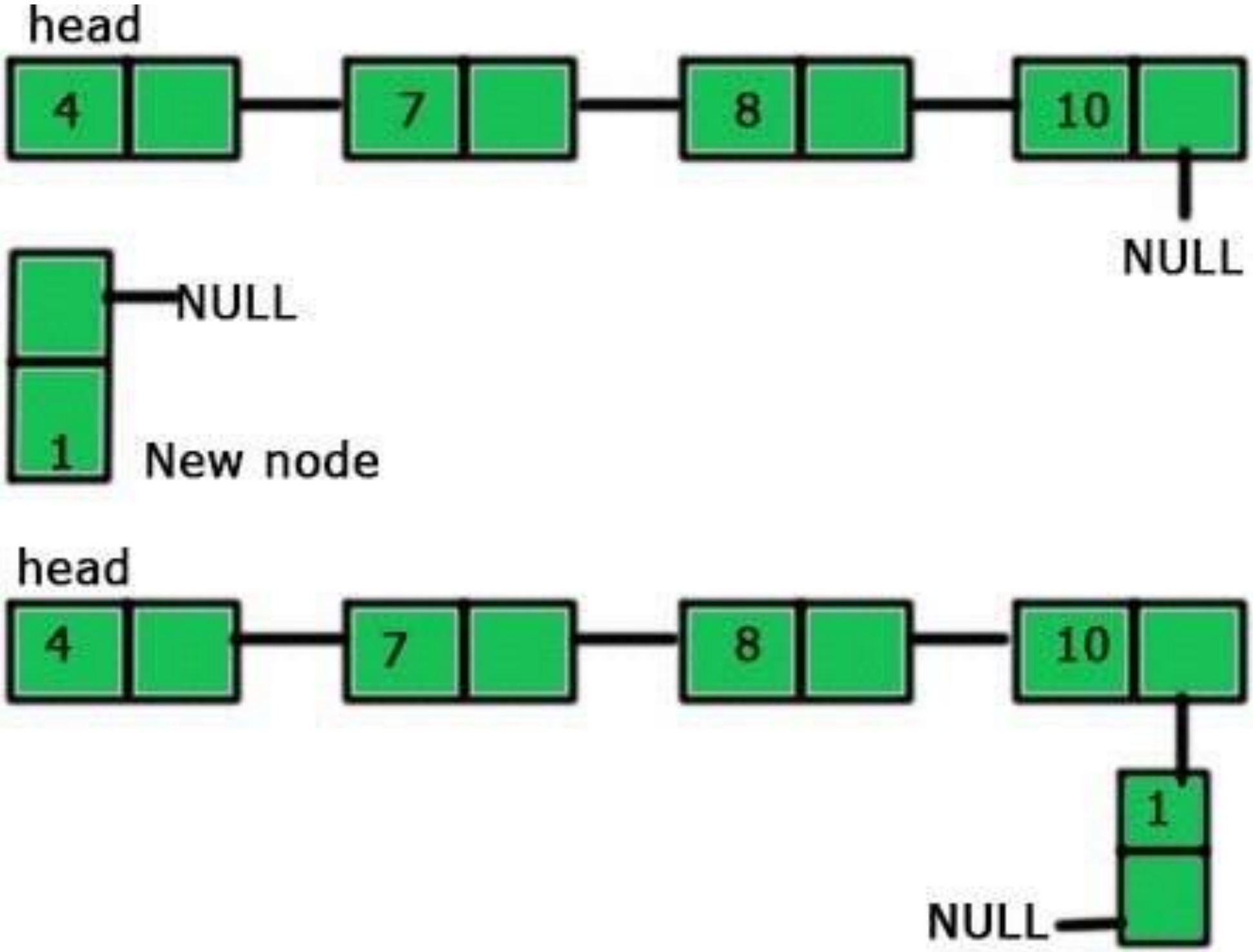
Insertion at the beginning of the linked list

Assume that the node x is created using CREATE_NODE function

```
void LIST_INSERT_FRONT(LL L, node x)
{
        x->next = L->head;
        L->head = x;
}
```



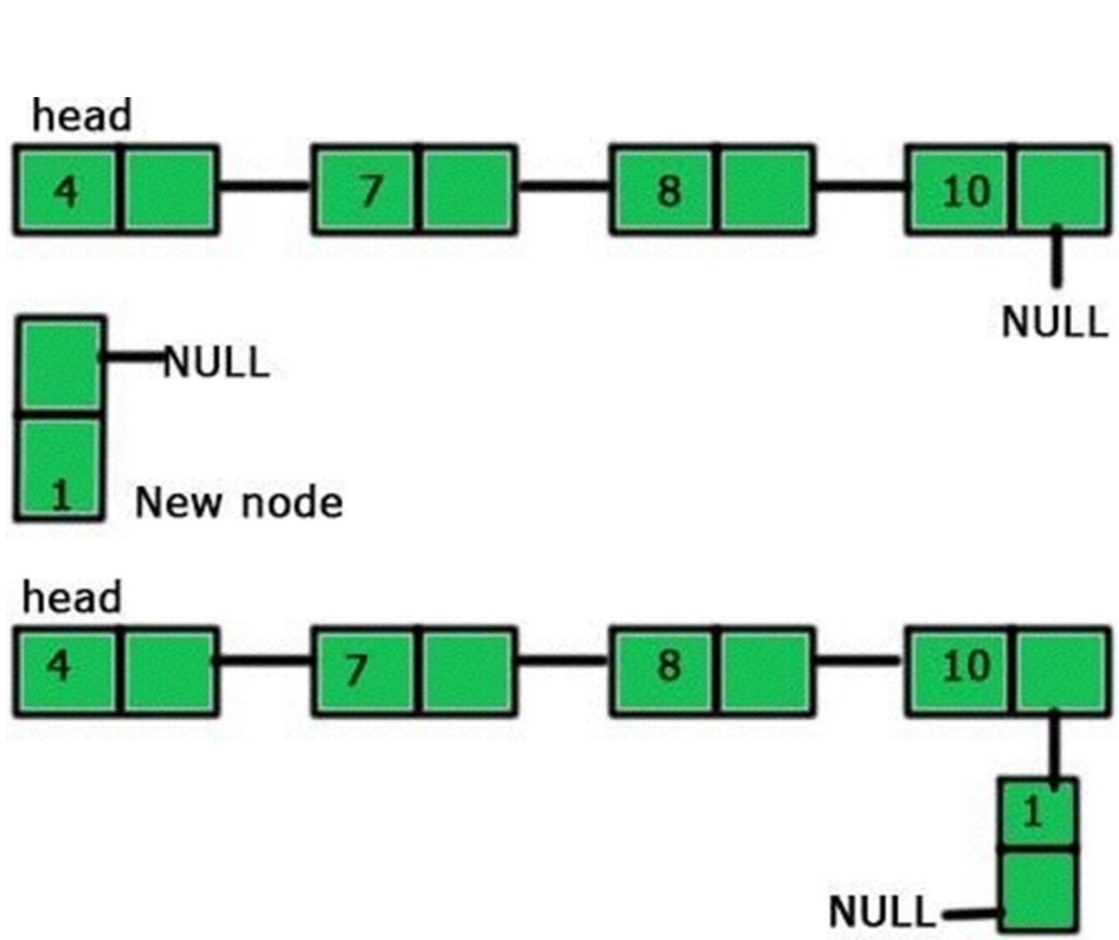
Insertion at the end of the linked list



Inserting a node to the end/tail of the linked list

Assume that the node x is created using CREATE_NODE function

```
void LIST_INSERT_TAIL(LL L,node x)
{
        node selected=L->head;
        if(selected!=NULL)
        {
                while(selected->next!=NULL)
                        selected=selected->next;
                selected->next=x;
        }
        else
                L->head=x;
}
```



Inserting a node to the end/tail/rear of the linked list

- Here the new node will always be added after the last node. This is known as inserting a node at the rear end.

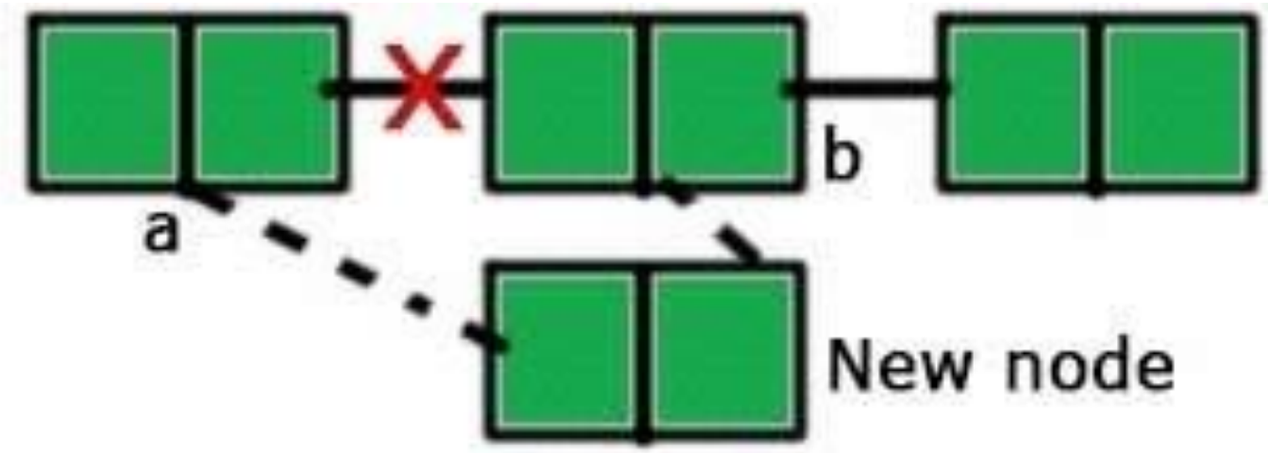
- A simple linked list can be traversed in only one direction from head to the last node.

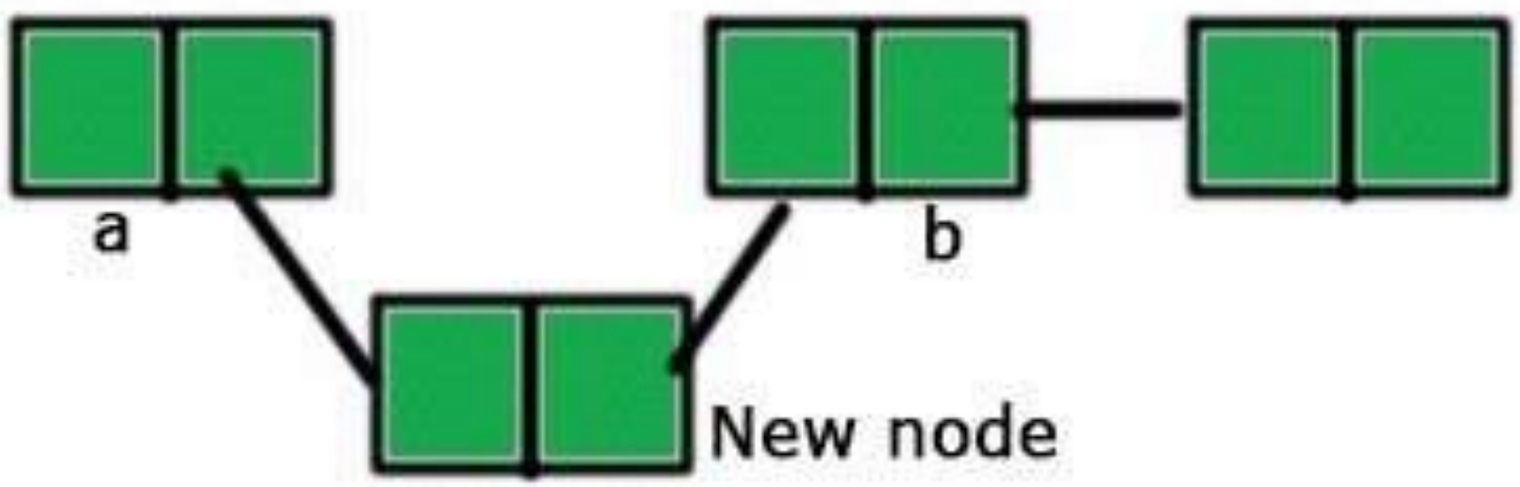
- -> is used to access next sub element of node p. NULL denotes no node exists after the current node, i.e. its the end of the list.

Insertion at the end of the linked list

What is the running time to insert the element in the tail of the list?

Insertion in-between the linked list





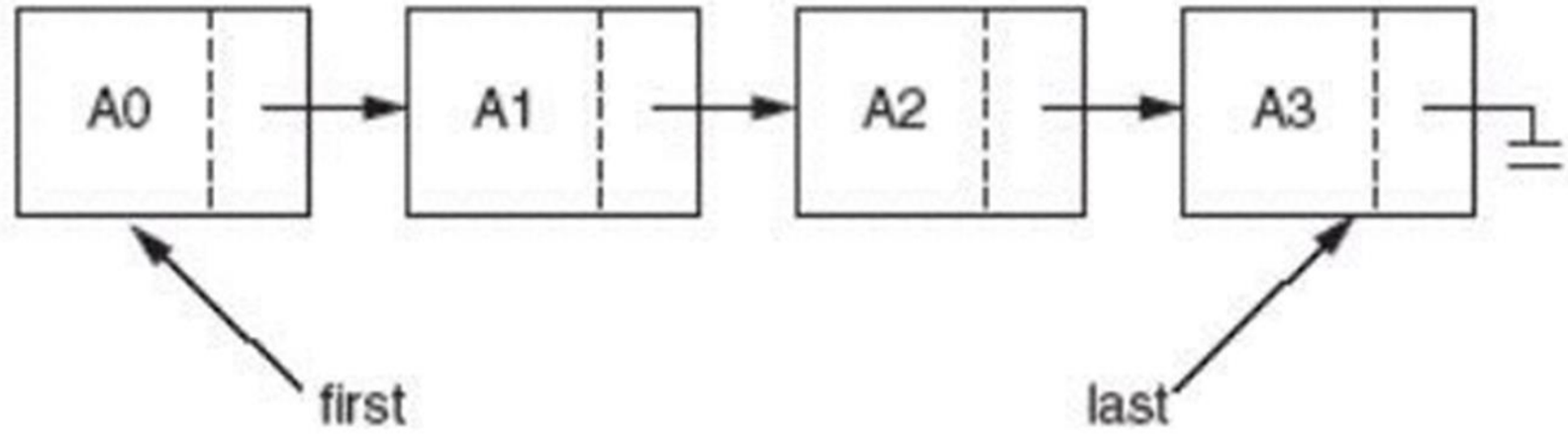
LIST-INSERT at a specific position EXERCISE

Write the Pseudocode/Algorithm to insert the element x at a particular position?

What is the running time to insert the element x at a particular position the list?

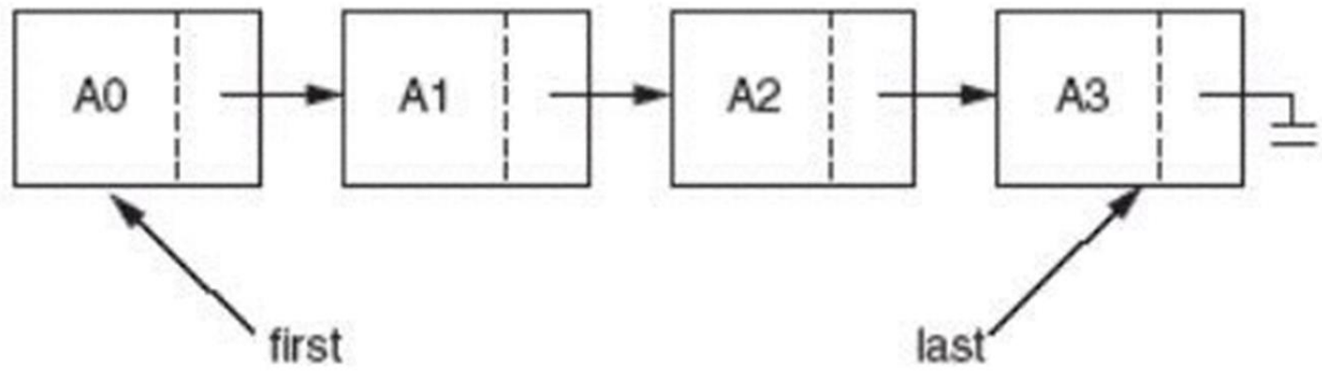
Search the LL for a key k

```
node LIST_SEARCH(LL L, long int k)
{
        node selected=L->head;
        while(selected!=NULL && selected->key!=k)
                selected=selected->next;
        return selected;
}
```



Printing the linked list

```
void print(LL L)
{
    node selected=L->head;
    while(selected!=NULL)
    {
        printf("%ld-->",selected->key);
        selected=selected->next;
    }
    printf("\n");
}
```



"The best way to learn a new programming language is by writing programs in it."

- Dennis Ritchie