Linked Lists

Agenda

- Dynamic Sets
- Operations on Dynamic Sets
- Representation of Dynamic sets
- Linked List - Introduction
- Linked List Operation (Search)

Dynamic Sets

- Sets are fundamental to **Mathematics & Computer Science(CS)**
- **Mathematics** - sets are unchanging
- CS sets are manipulated by algorithms
- Sets can
  - grow
  - shrink
  - change their size over time
- Such sets are called **Dynamic Sets**

Dictionary

- **Operations on dynamic sets**
  - Insert elements
  - Delete elements
  - Test membership

- **Dictionary -** Dynamic set that supports these operations

Totally Ordered Set

- **Examples:** Real numbers/natural numbers, alphabetic order of names

- Satisfies **Trichotomy property:** For any two elements **a and b** in the Totally Ordered Set, exactly one of the following must hold:
  - $a < b$
  - $a = b$
  - $a > b$

- **Totally ordered set**
  - Minimum/Maximum element of the set
  - Next element larger than a given element

Operations on Dynamic Sets

- Grouped into **two categories**:
  - **Queries**: return information on the sets
  - **Modifying operations**: change the set
- Depending on the application only few operations needed

Typical Queries on Dynamic Sets

- **SEARCH(S, k)**

  Input: A set *S* and a key value *k*

  Output: Returns

  – a pointer *x* to an element in *S* such that *x.key = k*

  – NIL if no such element belongs to S

- **MINIMUM(S):**

  Input: A totally ordered set *S*

  Output:

  – Returns a pointer to the element of S with the smallest key

Typical Queries on Dynamic Sets

- **MAXIMUM(S)**

  Input: A totally ordered set *S*

  Output:

  – Returns a pointer to the element of S with the largest key

- **SUCCESSOR(S, x)**

  Input: An element *x* whose key is from a totally ordered set *S*

  Output: Returns

  – a pointer to the next larger element in *S*

  – NIL if *x* is the maximum element in *S*

Typical Queries on Dynamic Sets

- **PREDECESSOR(S,x)**

  Input: An element *x* whose key is from a totally ordered set *S*

  Output: Returns
  - a pointer to the next smaller element in S
  - NIL if x is the minimum element in S

- **SUCCESSOR(S, x) and PREDECESSOR(S,x)** are extended to sets with non-distinct keys

Modifying operations on Dynamic Sets

- **INSERT(S,x)**
  Input: A set *S* and a pointer to *x* (Assume that an attribute of x is already initialized)
  Output:
  – Augments S with the element pointed by *x*

- **DELETE(S,x)**
  Input: A set *S* and a pointer to *x* (not a key value)
  Output:
  – Removes x from S

Measuring Running time for the operations on Dynamic Sets

- How do we measure the time taken to execute a set operation?
- In terms of the size of the set

Elementary Data Structures

- Representation of dynamic sets by simple data structures that use pointers :
  - Linked Lists
  - Stacks
  - Queues
  - Rooted Trees

Dynamic Sets

- Each element is represented by an object
  - A pointer to the object is used for examining and manipulating the objects' attributes

- Dynamic sets assume, one of the objects attribute is an identifying **key**

Dynamic Sets

- The object may contain **satellite data**, which are carried around in other object attributes
- **Object attributes** - manipulated by set operations
  - Attributes may contain data or pointers to other objects in the set
- Some dynamic sets - **keys from a totally ordered set**

Linear Data Structures

- Array - order determined by the indices
- Advantage
- Disadvantage

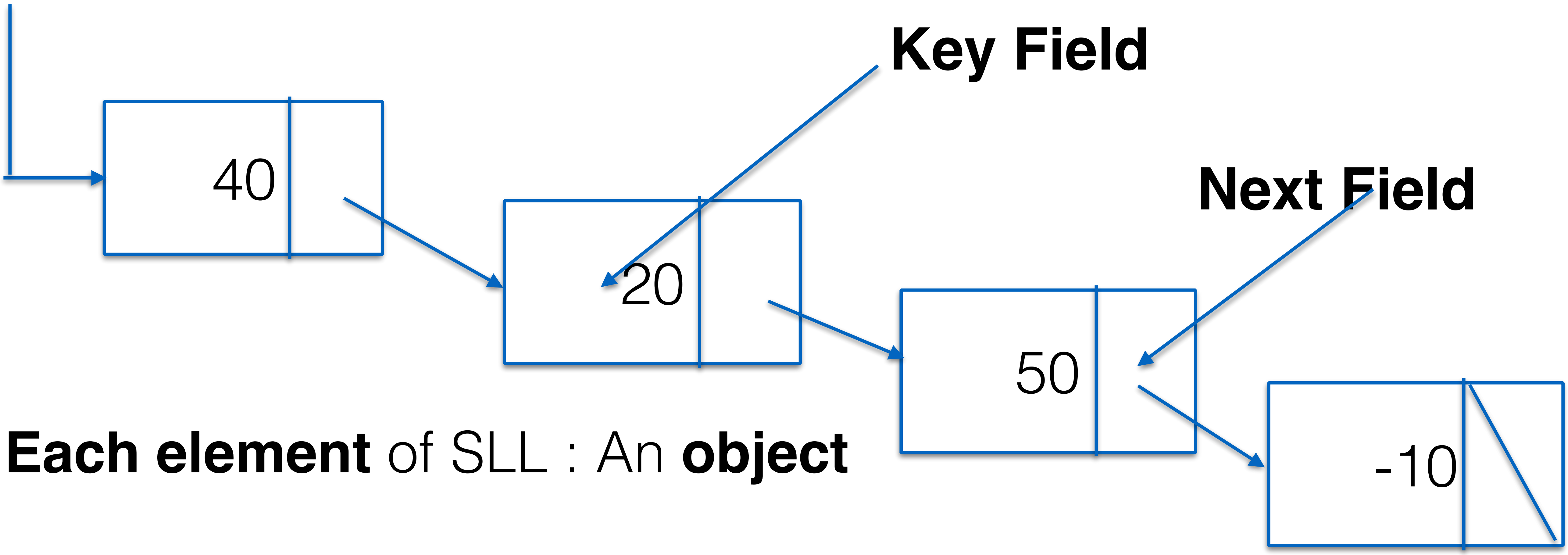
Linked Lists

- **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)

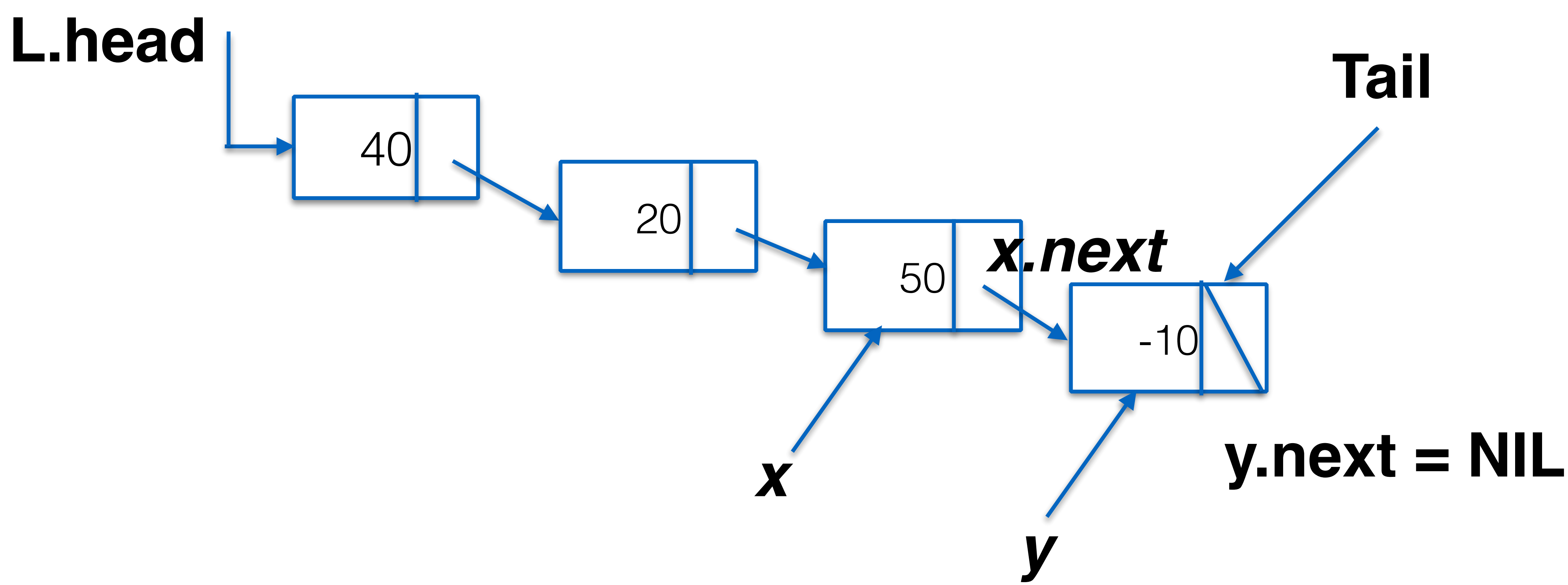


Each element of SLL : An **object**

Attributes: Key and a Next pointer

Object may also contain other **satellite data**

SINGLY LINKED LIST (SLL)



- An attribute **L.head** points to the **first element** of the list. If **L.head = NIL, the list is empty**
- Given an element *x* in the list, ***x.next*** points to its **successor** in the linked list
- If **x.next = NIL**, the element x has **no successor** and is therefore the last element, or **tail**, of the list.

Types of Linked List

- **Sorted List** - If a **list is sorted**, the **linear order** of the list corresponds to the **linear order of keys** stored in elements of the list
  - **Minimum** element: head of the list
  - **Maximum** element: tail of the list
- **Unsorted List** - If the list is **unsorted**, the elements can **appear in any order**.

Search Operation

The procedure **LIST-SEARCH (L,k)** finds the first element with **key k** in **list L** by a simple linear search, returning a pointer to this element.

If **no object with key k** appears in the list, then the procedure **returns NIL**.

LIST-SEARCH (L,k)

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

LIST-SEARCH (L,k)

Consider a SLL with **n objects**, What is the running time of LIST-SEARCH ?

- Best Case
- Worst Case
- Average Case

Insertion of a node in a linked list

1. Insertion at the beginning of the list.

2. Insertion at the end of the list

3. Inserting a new node except the above-mentioned positions.

LIST-INSERT (L,x)

Steps to insert the element x in the front of the SLL:

1. x.next = L.head

2. L.head = x

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

O(1)

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

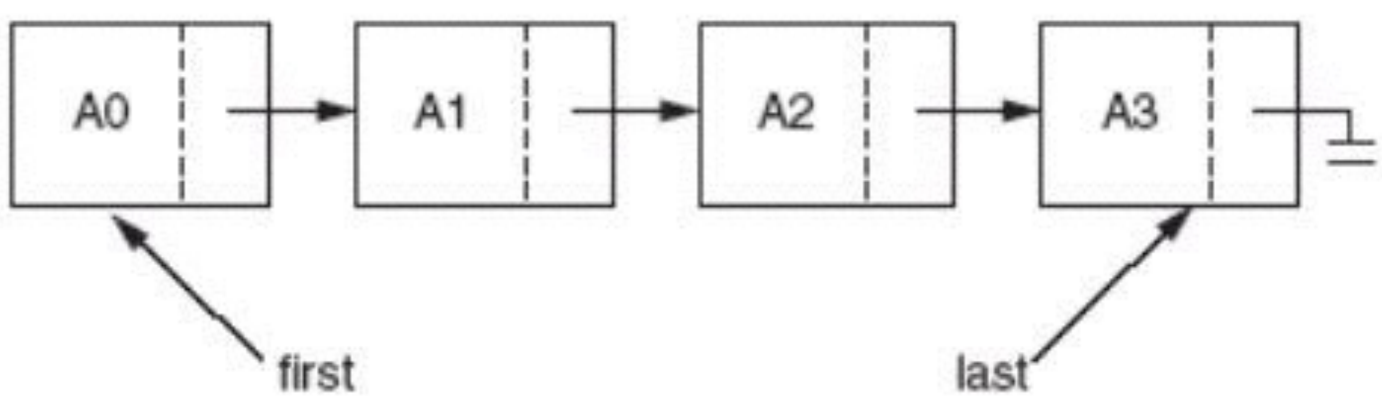
- 4 DYNAMIC MEMORY ALLOCATION FUNCTIONS IN C:
  - malloc()
  - calloc()
  - realloc()
  - free()

Linked List

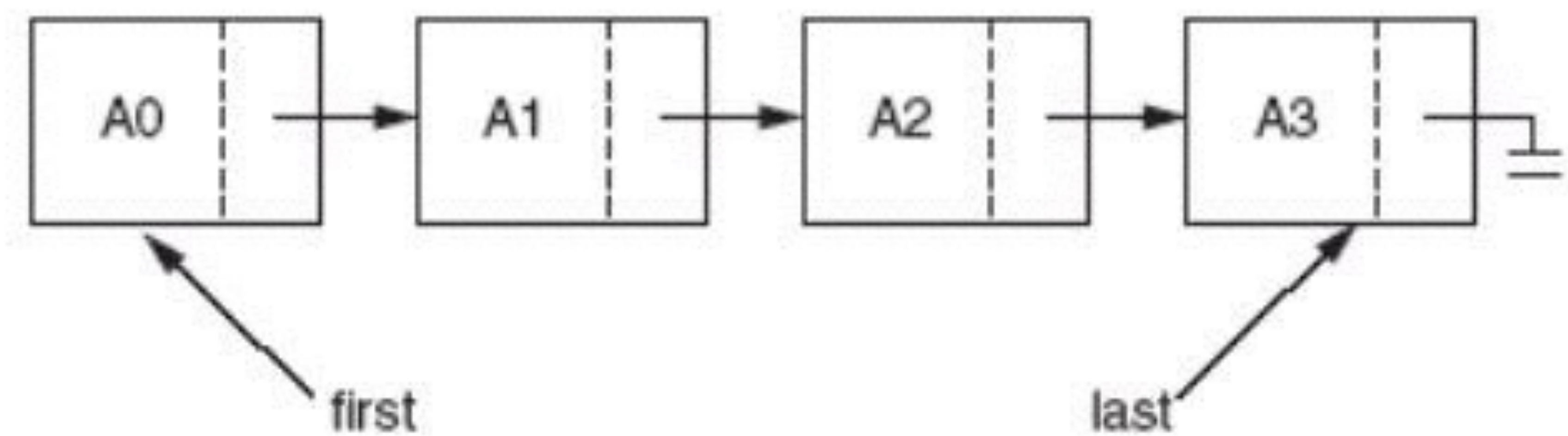
- A linked list is simply a chain of structures which contain a pointer to the next element and it is dynamic in nature.

- 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 Linked list

Declaring a Linked list :

```
struct LinkedList{
    int data;
    struct LinkedList *next;
 };
```

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

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

In place of a data type, **struct LinkedList** 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 and it will point to the next node.

Creating a Node

```
typedef struct LinkedList *node;
                                       //Define node as pointer of data type struct LinkedList
node createNode() {
    node temp;                          // declare a node
    temp = (node ) malloc (sizeof(struct LinkedList));
                                        // allocate memory using malloc()
     if(temp == NULL)
    {   printf("Error creating a new node.\n");
        exit(0);   }
    temp->next = NULL;   // make next point to NULL
    return temp;                        //return the new node
}
```

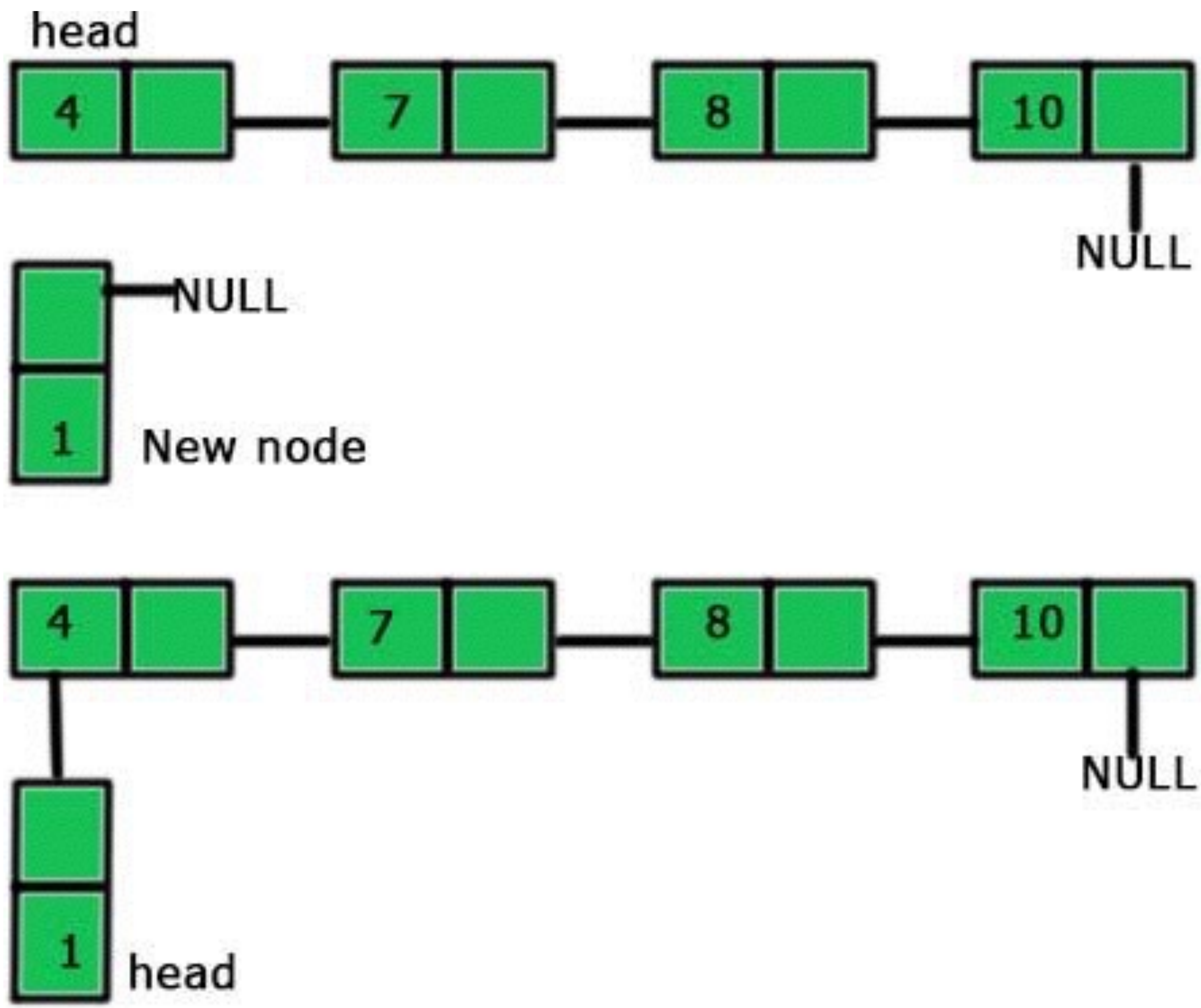
typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

The above code will create a node with the next field pointing to NULL.

Insertion at the beginning of the linked list



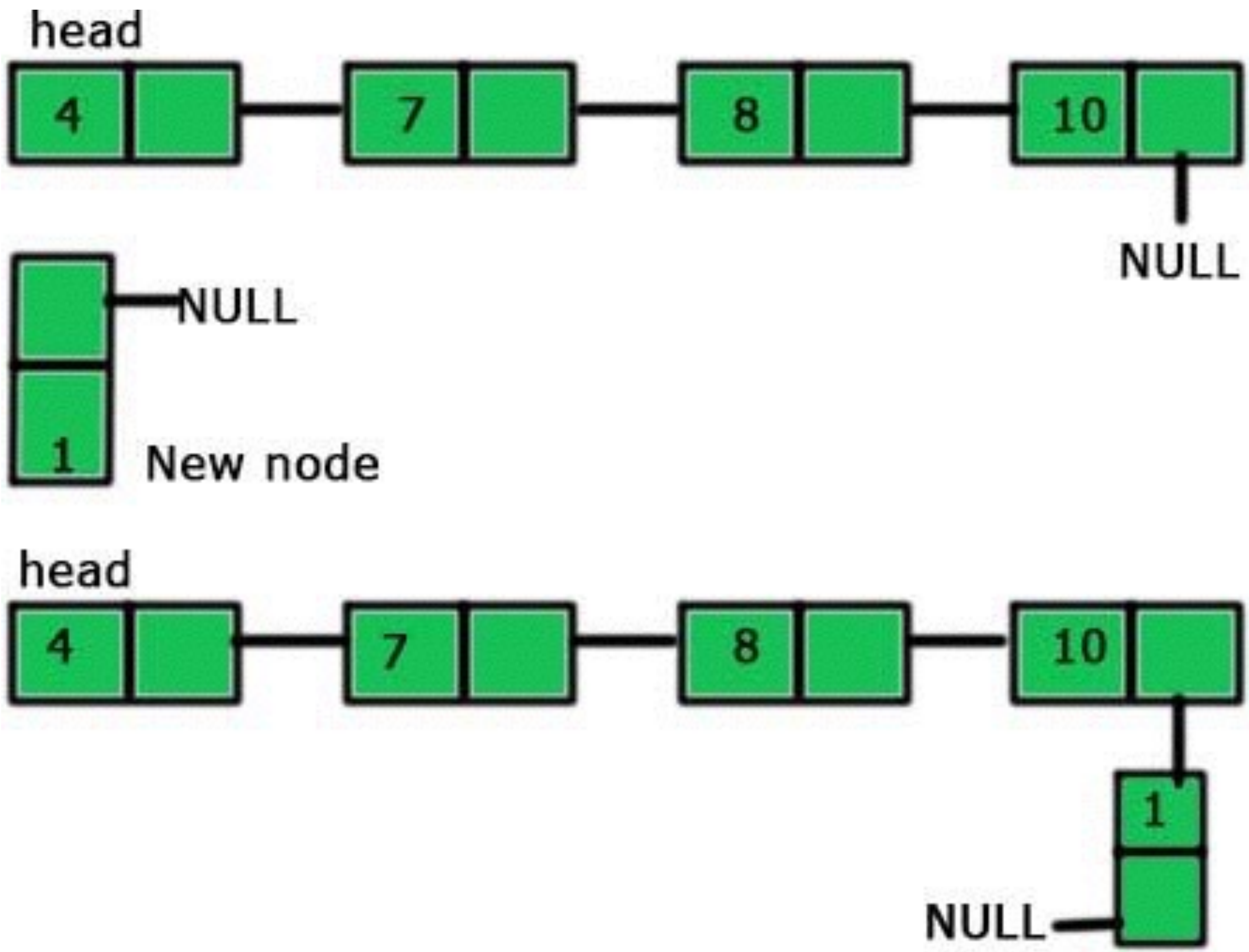
Insertion at the beginning of the linked list

```
node insert-at-begin(int data, node first)
{
   node  new_head;

   node new_node =createNode();
   new_node->data = data;
   new_node->next = first;

   new_head = new_node;
   return new_head;
}
```

How do we call the above function?

```
node head;
head = insert_at_begin(10, head);
```

Insertion at the end of the linked list



Inserting a node to the end of the linked list

```
node addNode(node head, int value) {
  node temp,p;                     // declare two nodes temp and p
  temp = createNode();             //createNode will return a new node with data
                                   // = value and next pointing to N ULL.
  temp->data = value;              // add element's value to data part of node
  if(head == NULL) {   head = temp;   }    //when linked list is empty
  else {
    p  = head;              //assign head to p
    while(p->next != NULL)
      p = p->next;      //traverse the list until p is the last node.The last
                        // node always points to NULL.
    p->next = temp;  //Point the previous last node to the new node created.
  }
  return head;
}
```

Inserting a node to the end 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.

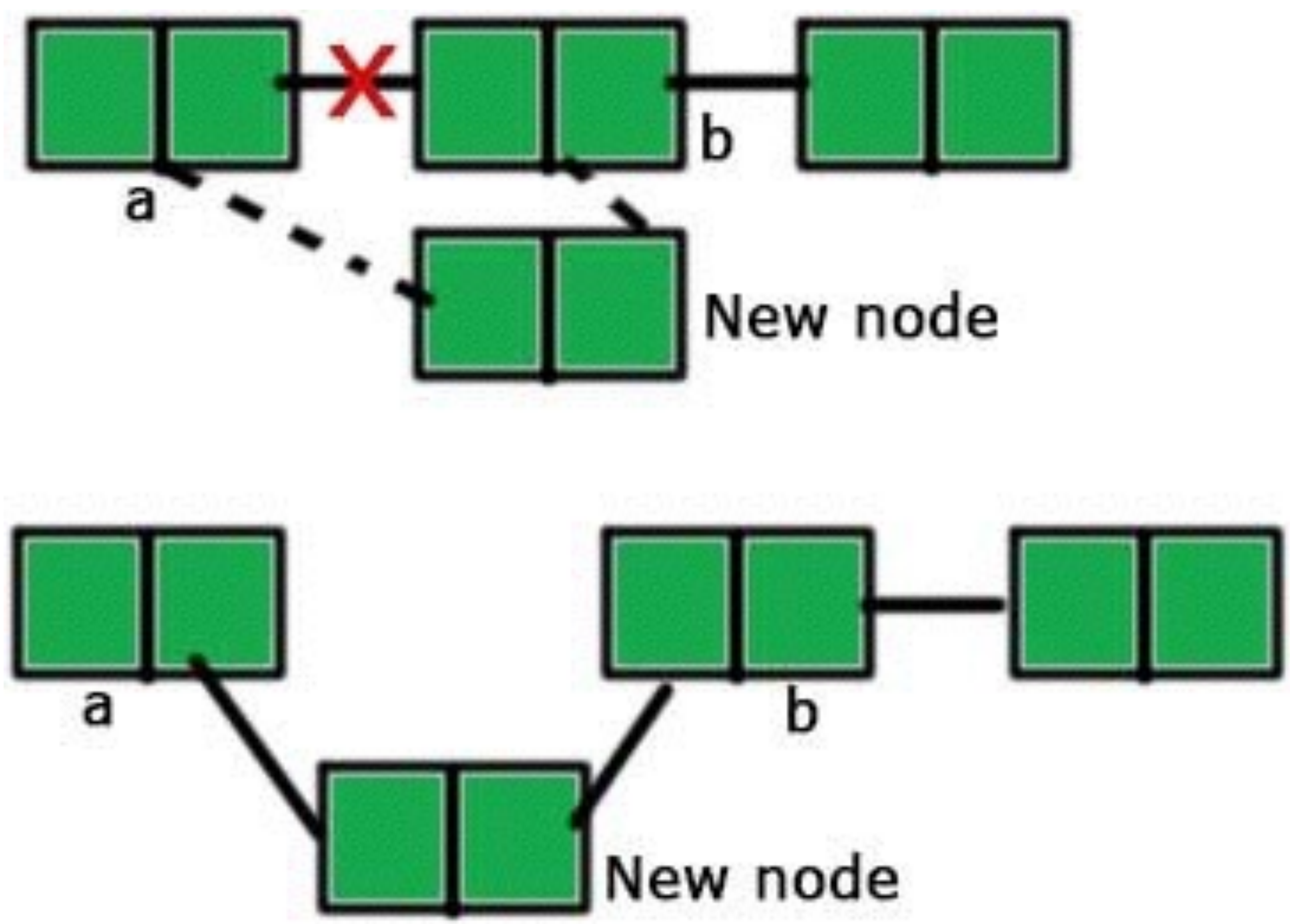
- A simple linked list can be traversed in only one direction from head to the last node.

- -> 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?

Traversing the list

The linked list can be traversed in a while loop by using the head node as a starting reference:

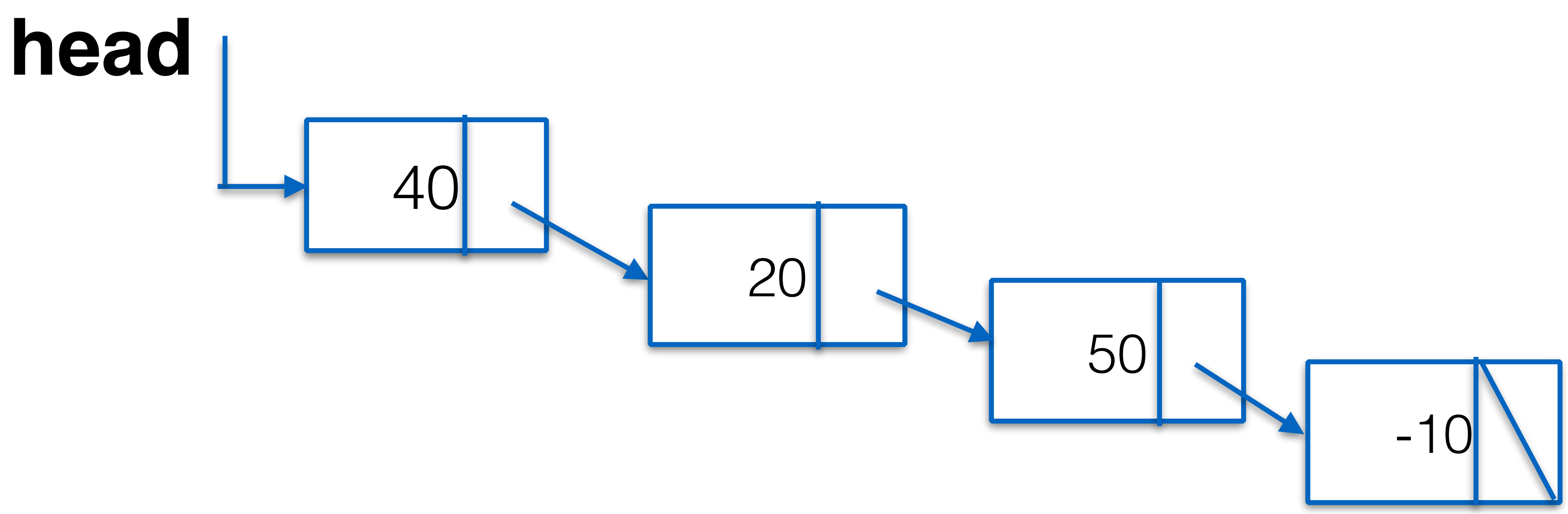
head is a pointer of type node.

```
node p;

p = head;

while(p != NULL)
    p = p->next;
```

Deletion of a node from a Linked List

- Delete a node from the front of the linked list
- Delete a node from the rear of the linked list
- Delete a node from the middle of the linked list
- Deleting the whole linked list

Deletion of a node: front of the Linked List



node x = head;

head = head -> next

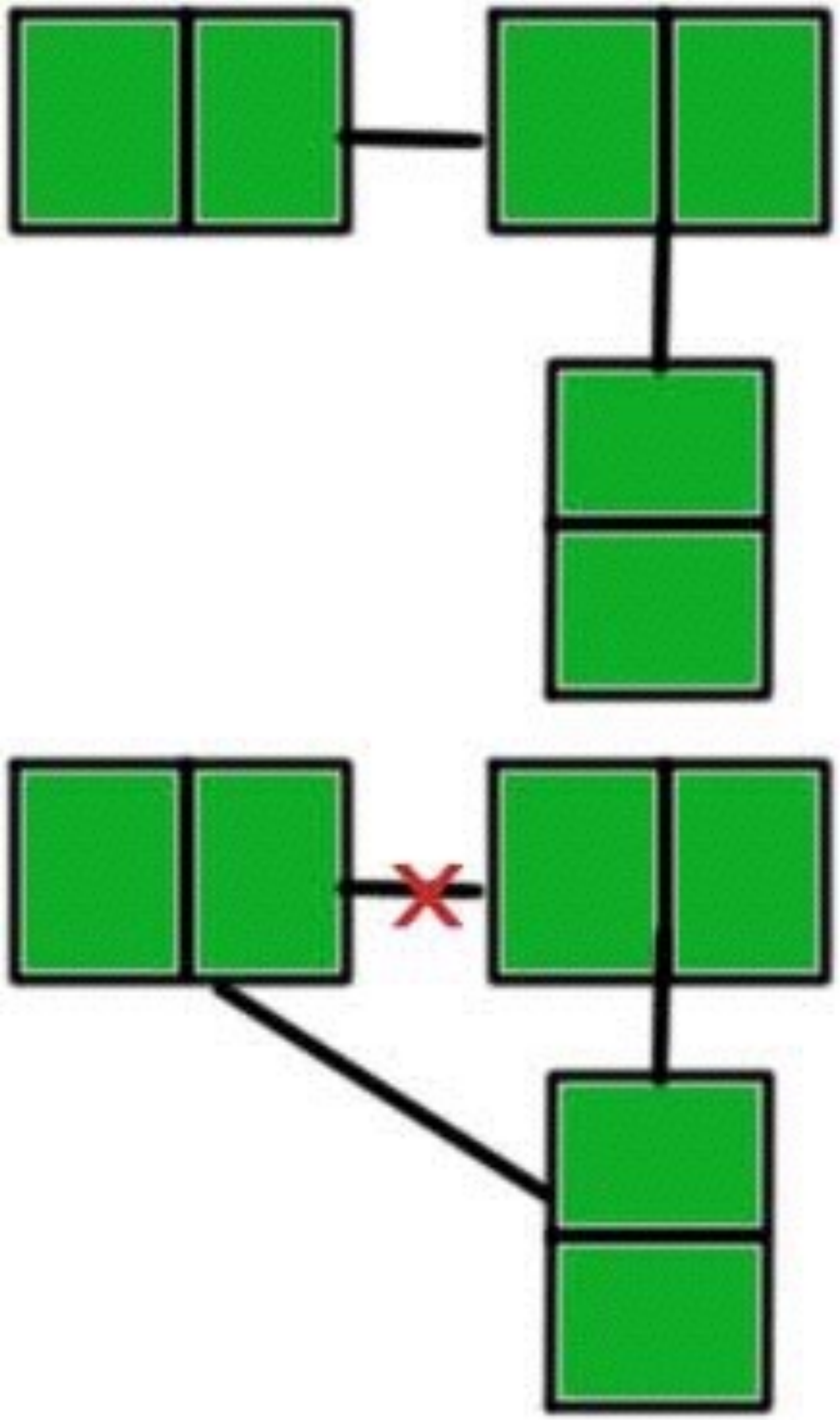
free x;

return head;

Delete a node from the rear of the linked list

```
node remove_tail(node head)
{   // check whether head  of the Linked list is null
    if(head == NULL)      return NULL;

   // two pointers to keep track of the current and the previous node
    node cursor = head; node prev= NULL;

    // Traversing the list
    while(cursor->next != NULL) {
        prev = cursor;
        cursor = cursor->next; }

    // Checking whether the node prior to the last node is not null
    if(prev != NULL)    prev ->next = NULL;

    // if this is the last node in the list
    if(cursor == head)
        head = NULL;

    free(cursor);

    return head;
}
```

Deletion of a node from the middle of the linked list



Delete the whole linked list

```
void dispose(node head)
{
    node cursor, tmp;

    if(head != NULL)
    {
        cursor = head;

        while(cursor != NULL)
        {
            tmp = cursor->next;
            free(cursor);
            cursor = tmp;
        }
    }
}
```

Exercises

- Count the number of nodes in a linked list
- Reversing the linked list

Linked list operations - Exercises

Insertion of a node (to maintain an ordered linked list)

- **Input:** Linked List and an element to be inserted
- **Output:** Ordered linked list

Deletion of a given node in the linked list

- **Input:** Linked list and an element x to be deleted
- **Output:** Ordered linked list without x

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

- Dennis Ritchie