Chapter 2
Lists

Definition and types

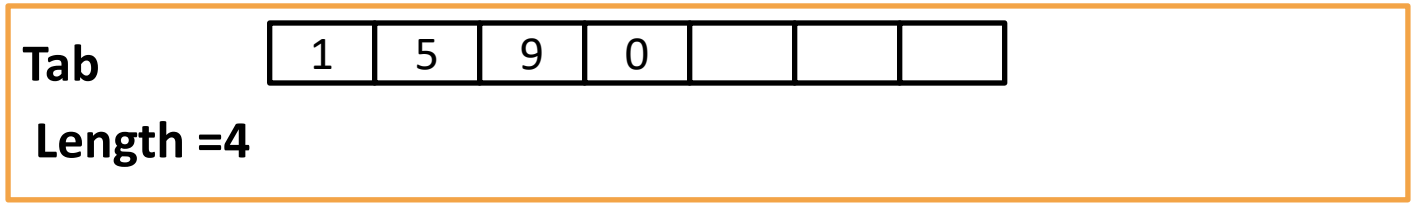
Definition: A list is a finite (possibly empty) sequence of elements of the same type identified according to their rank in the list.

Example: the list of integers L = {1,5,9,0}

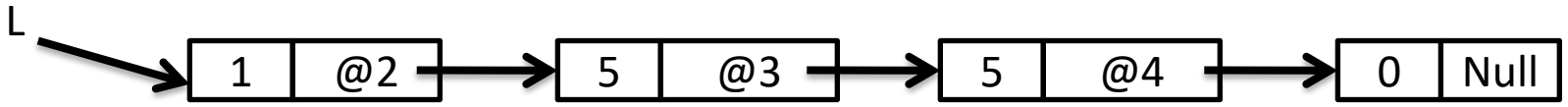
the rank of **5** is **2** .

Types: two types

- **Static list (contiguous):** Represented by an array **Tab** of **elements** and a **length variable**:



- **Dynamic list:** representation by **cells** or **nodes:**



Static lists

Type List definition

Type Structure List

Begin

Tab : Array[MAX] **of Elements;**

length : Integer ;// keeps the number of elements stored in the list

End structure;

Example: list of integers

Type Structure ListIntegers

Begin

Tab: Array[1000] of integers;

length: Integer;

End

Static lists

Advantages

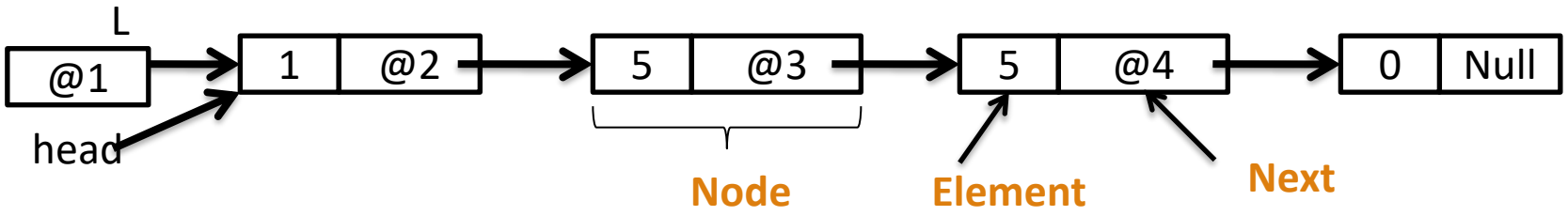
- Easy route and access to the ith element (direct access).
- Ability to search efficiently if the list is sorted (e.g. dichotomous search).

Disadvantages

- Reservation, during compilation, of the imposed Maximum Size (limited), which lacks flexibility and economy.
- Inefficiency of deletion and insertion inside the list: **obligation to shift all elements between the inserted or deleted element and the last element.**

Linked lists

- A linear linked list is a set of dynamically allocated **Nodes** linked together.



- A **Node** is a record with two fields:
- **Element field** (Ele) containing the information.
- **Next field** giving the address of the next Node.

- The address of the first element of a linked list is often called the head of the list .

- **Null** ***(NULL** in **C)** is used to indicate the end* of the list in the last link.

Declaration

a) **List type definition**

Type Structure Node

Ele : typeq ; // typeq designates any type (int, float , person, student...etc).

next: * Node ;

END

Type List : * Node; // The list type designates all pointers to a Node

- **Use for declaration**

L: **List** ; <==> L: *** Node ;**

L, P, Q: List // means that L, P, Q are pointers to Nodes

Declaration

b) List definition in C++

```
structure Node
{
   Typeq Ele ;
   Nodes *next;
};
typedef Node * List;
```

Use for declaration

List L, Q, head ; equivalent to **Node * L, Q, head ;**

Linked lists of integers

Example 1: linked list of integers

```
Type Structure Node
    Ele : integer;
    next: * Node;
End structure;
Type List : * Node;
```

Chained lists of people

Example 2 : linked list of people

```
Type Structure Person
    Lastname: character string
    Firstname: character string
    Age: Integer
END
Type structure Node
    Ele : person;
    next: * Node ;
END
Type List : * Node;
```

Chained lists of students

Example 2 : Linked list of students

```
Type Student structure
        P: Person
        Num: integer
END
Type Structure Node
    Ele : Student;
    Next : * Node;
END
Type List: * Node;
```

Or

```
Type Student_List : *Node
```

Primitive operations

- **Create_Node (X: element)**
- **Is_empty (L: List)**
- **Length (L: list)**
- **Add(var L: List, x: Typeq)**
- **First (L: list)**
- **Rest (L: list)**

Linked Lists (Primitive Operations)

1. **Create_node** : creates a new Node, with dynamic memory reservation (Allocate), containing the value **x** in the field element and **Null** in the Next field. This function returns the address of the created Node.

Function Create_Node (x: typeq) : List

P: List // or P: * Node

Begin

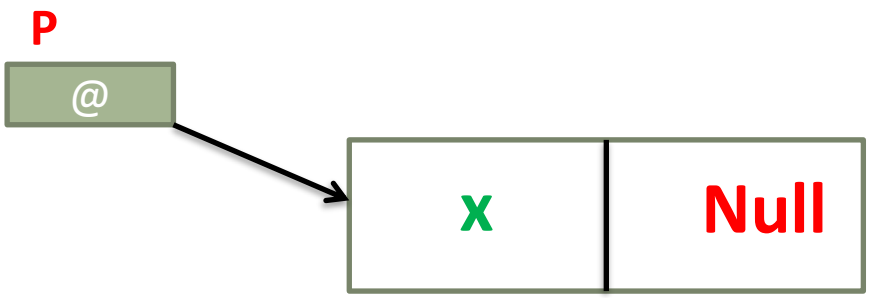
P ← **Allocate (Node)**

P - > Ele ←x;

P -> next ← Null;

Return (P);

END

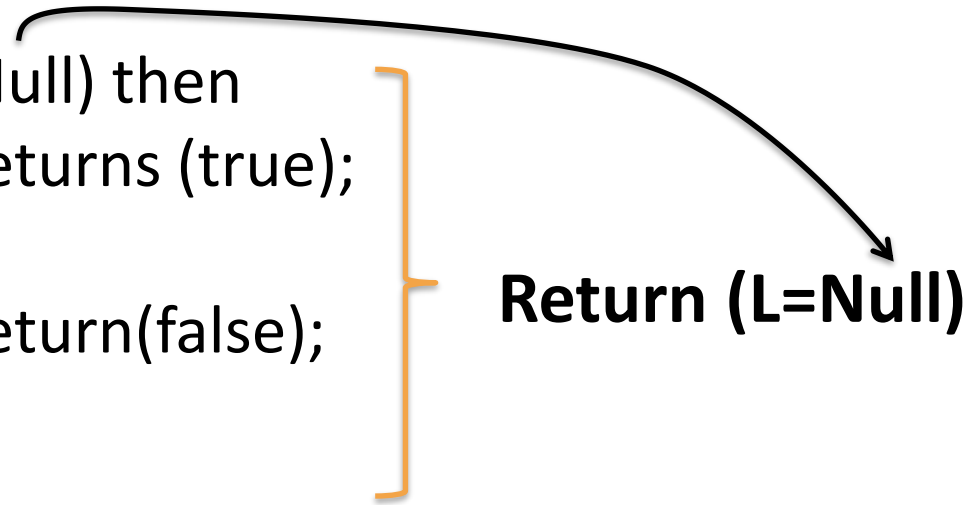


Linked Lists (Primitive Operations)

2. Is_empty : tests if the list is empty or not

Function Is_empty(L: List): Boolean

```
Begin
    If (L=Null) then
          Returns (true);
    Else
          Return(false);
    End if
END
```



Linked lists (Primitive operations)

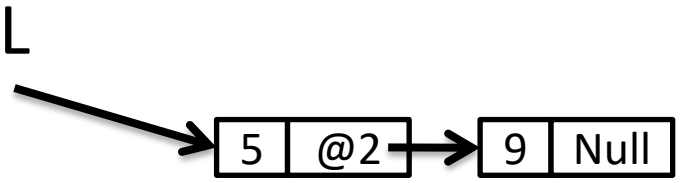
3. First: returns the first element of the list L

Function First (l: list): Element,

Begin

returns (L **->** Ele);

END



L -> ELE = 5

Examples of use

write (First (l));

x ← First (L);

Linked lists (Primitive operations)

4. Add : inserts a new element at the head of the list L. It can be written as a function, returning the new list , or as a procedure with a passage by variable in the list.

Function Add(L: List, x: typeq): List

P: List // or P: * Link

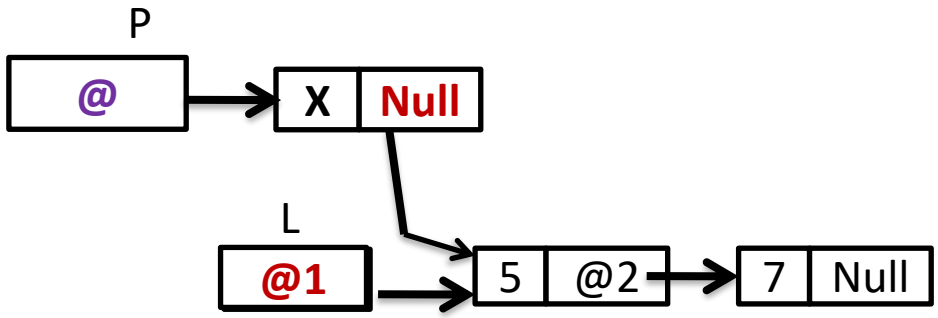
Begin

P ← Create_Node (x);

P -> next ← L;

Returns (P);

END



Linked lists (Primitive operations)

Procedure Add (var L: List, x: Element)

P: List

Begin

P ← Create_Node (x);

P -> next ← L;

L ← P ;

END

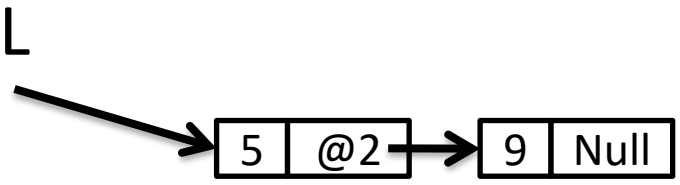
Linked lists (Primitive operations)

5. Rest: returns the list L without the first element. It returns the address of the second node in the list (returns the value of the next field of the first node).

```
Function Rest (L: List): List

Begin
    Return ( L -> next );
END
```

Examples of use

```
write (first (Rest(L));
```



prime (Rest(L)) = 9

Linked lists (Primitive operations)

6. Length: returns the number of elements in a list L.

Iterative

```
Function Length(L: List): integer
    current: List
    Nb: integer
Begin
    Number ←0; current←L;
 While (current ≠ Null) do
    Nb ← Nb+1;
    current ← current ->next.
 End while
    return Nb;
END
```

Linked lists (Primitive operations)

6. Length: returns the number of elements in list L.

Recursive version:

```
Function Length(L: List): integer

Begin
    If (L=Null) then
            Return (0);
    else
            Return 1 + length(Rest (L)) ; // recursive call
    End if
END
```

Linked lists (Other operations)

Display : allowing you to display the elements of a list L.

Iterative version

```
Procedure display (L: List)
    current: List;
    Nb: integer;
Begin
    current ←L ;
 While (current != Null ) do
    write ( current -> Ele );
    current ← current ->next.
End while
END
```

Linked lists (Other operations)

Show: allowing you to display the elements of list L.

Recursive version

Procedure Display (L: List) // recursive version.

Begin

If (L!=Null)

  write (first (L))

  Display (Rest(L)) // recursive call

End if

END

Linked lists (Other operations)

Last: returns the last element of list L. The list must contain at least one element.

Iterative version

```
Function Last (L: List): item

Current: List

Begin
    Current ←L ;
 While (current->next ≠ Null ) do
    current ← current ->next.
 End while
    return ( current -> Ele );
END
```

Linked lists (Other operations)

Last: returns the last element of list L. The list must contain at least one element.

Recursive version

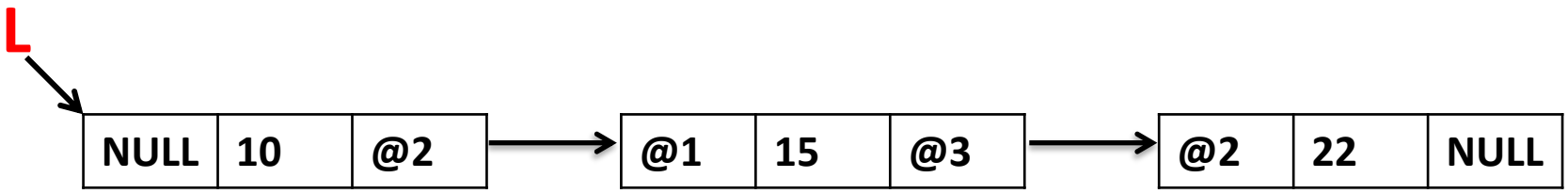
```
Function Last (L: List): item

Begin
If ( is_empty (rest (L))) then
   returns (first (L))
Else
   returns Last (Rest(L)) // recursive call
End if
END
```

Doubly Linked list

- A doubly linked list is a more complex data structure than a singly linked list.
- Each node in the list contains a pointer to the previous node and a pointer to the next node.
- The main advantage is that it allows for efficient traversal of the list in both directions and easy insertion and deletion of nodes from the list.

Doubly Linked list

- A doubly linked list is represented using nodes that have three fields:
  - Ele: containing the list element or the value.
  - A pointer to the next node (**Next**).
  - A pointer to the previous node (**Prev**).



Doubly Linked list

Type definition

```
Type Structure Node
    Ele : typeq ; // typeq designates any type (int, float , person, student...etc).
    Next: * Node ;
    Pre   : * Node ;
END
```

Doubly Linked Lists

1. **Create_node** : creates a new Node, with dynamic memory reservation (Allocate), containing the value **x** in the field element and **Null** in the Next and Prev fields. This function returns the address of the created Node.

Function Create_Node (x: typeq) : List

P: List // or P: * Node

Begin

P ← **Allocate (Node)**

P - > Ele ←x;

P -> next ← Null;

P-Prev ←Null;

Return (P);

END

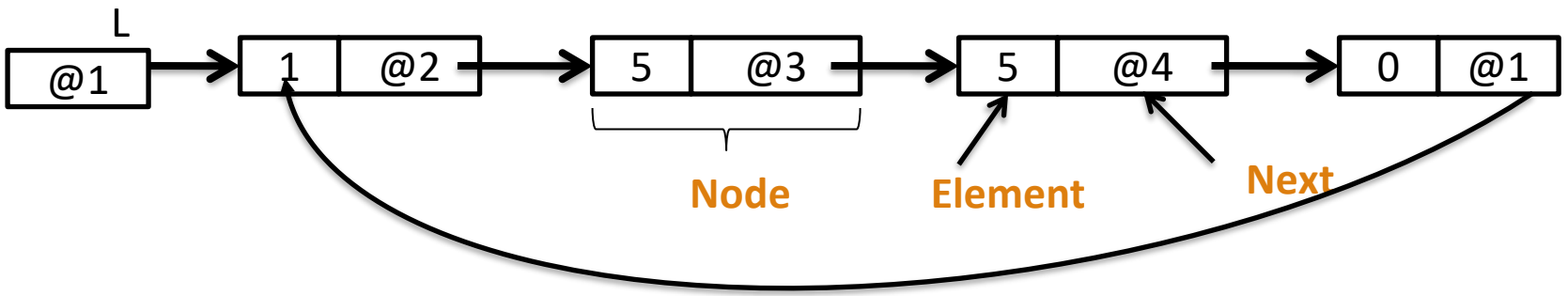
Doubly Linked lists

Procedure Add (var L: List, x: Element)

```
P: List

Begin
   P ← Create_Node  (x);
   P -> next ← L;
   if L ≠ NULL then
         L->prev ←P;
    end if
   L ← P ;
END
```

Circular Linked list

- Is a data structure where the last node connects back to the first, forming a loop. This means that you can keep traversing the list without ever reaching a **NULL** value



- This structure allows for continuous traversal without any interruptions.

- Circular linked lists are especially helpful for tasks like **scheduling** and **managing playlists**.