Chapter 3. Linked Lists

Dr. Abderrahmane Kefali

Senior Lecturer Class A,

Department of Computer Science,

University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

All variables manipulated so far, including those of complex types declared prior to program execution, are considered ***static*** variables. This implies that the memory space allocated for them is precisely determined during program compilation and remains constant throughout execution.

However, algorithmics offers us the possibility to define various other types of advanced data structures, called ***dynamic data structures***, mainly using the concept of records. Variables of these types can be created during program execution, and their size can evolve during program execution. These types essentially include ***linked lists***, ***stacks***, ***queues***, etc.

In this chapter, we focus on the study of linked lists, considered as the basic model of other dynamic structures. Indeed, linked lists are closely related to two other very important concepts: ***pointers*** and ***dynamic memory allocation***. It is therefore essential to address these two concepts first before presenting linked lists.

2) Pointers

The central memory of a computer consists of numerous fixed-size (in bytes) slots called ***memory words***. Each word is identified by a unique number known as the ***address*** of the memory word or slot. In algorithmics and the C language, address manipulation is facilitated through variables known as ***pointers***.

2.1) Notion of Address

Ainsi, la zone mémoire occupée par une variable est accessible à travers son identificateur ou son adresse. On peut connaître l'adresse d'une variable en faisant procéder le nom de la variable par:

During the compilation of a program, each declared variable occupies contiguous memory slots. The address of the variable is the address of its first memory slot. This type of variable is called ***static variables***, and the necessary memory space is reserved at the beginning of the program execution.

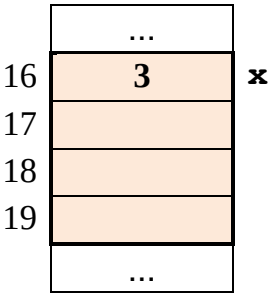
Thus, the memory area occupied by a variable is accessible through its identifier or its address. We can obtain the address of a variable by preceding the variable name with:

- In algorithmics : the operator: "**@**".
- In C language: the operateur "**&**".

Example:

```
Var x:integer;
.....
x ← 3;
write(@x);
```

```
int x;
....
x=3;
printf("%d",&x);
```



This code snippet displays 16, which is the address of the integer variable **x**, as illustrated in the schemas on the right.

2.2) Definition

A pointer is a variable designed to store a memory address (the address of another variable). The pointer is associated with a type of object, such as a pointer to an integer, a pointer to a float, a pointer to a character, or any other type (including a pointer). By accessing this address, we can indirectly access the variable and therefore modify it. Therefore, when a variable **p** of pointer type contains the address of another variable **x**, we say that **p** points to **x**. The variable **p** is the pointer, and **x** is the pointed variable.

2.3) Declaration

2.3.1) In algorithmics

In algorithmics, a pointer type is declared by preceding the pointed type with the character "**^**". The declaration syntax is as follows:

```
Type <pointer_type> = ^<pointed_type>;
```

Then, declare the pointer variable as usual:

```
Var <pointer_name>: <pointer_type>;
```

It is also possible to declare a pointer variable directly without defining a new type as follows:

```
Var <pointer_name>: ^<pointed_type>;
```

Examples:

Consider the following declarations:

1) `Var p:^Integer;`

   Declare a variable named **p** that is a pointer to an **integer**. The variable **p** is intended to store the address of an integer.

```
2)  Type  Student=Record
              Begin
              ID:Integer;
              lastName,firstName:String[20];
              End;
          ptrStudent=^Student;
    Var   stud:ptrStudent;
```

This declaration first defines a record type named **`Student`**. Then, it defines another type named **`ptrStudent`**, which is a pointer type to **`Student`**, and finally declares a pointer **`stud`** of this type.

Remarks:

1. The definition of a pointer allocates memory for storing a pointer but does not assign any value (address) to it.
2. It is possible to declare a pointer to a type that is not yet defined because the size of a memory location for a pointer is the same regardless of the pointed type (a pointer is typically coded on 4 bytes).

Example:

```
    Type  ptrStudent=^Student;
          Student=Record
              Begin
              ID:Integer;
              lastName,firstName:String[20];
              End;
    Var   stud:ptrStudent;
```

2.3.2) In C language

The declaration of a pointer type in C is done by adding an asterisk **`*`** after the pointed type. The syntax for the declaration is as follows:

```
typedef <pointed_type>* <pointer_type>;
```

Then, declare the pointer variable as usual:

```
<pointer_type> <pointer_name>;
```

The direct declaration of a pointer follows roughly the same syntax: add the asterisk **`*`** and specify the pointed type. The syntax for this declaration is as follows:

```
<pointed_type>* <pointer_name>;
```

Examples:

1) To declare a pointer to an integer named **`p`** directly, we write:

```
int* p;
```

2) The following lines declare a pointer to a **`Student`**. The latter is a record type.

```
typedef  struct {
          int ID;
          char lastName[20], firstName[20];
      }Student;
typedef Student* ptrStudent;
ptrStudent stud;
```

Remark:

As in algorithmics, it is possible to declare a pointer to a type that has not yet been defined. When the type of the pointed value is a structure, this type must be preceded by the keyword **`struct`** in the pointer declaration.

Example:

```
typedef  struct Student* ptrStudent;
typedef  struct {
          int ID;
          char lastName[20], firstName[20];
      }Student;
ptrStudent stud;
```

2.4) Initialization and Assignment

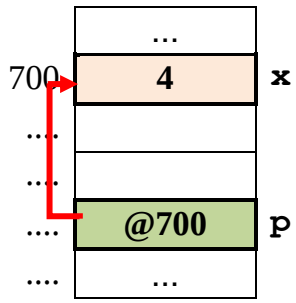
As with any variable, a pointer must be initialized before being manipulated. It must be initialized with the address of a variable having a type that matches that of the pointer. Assigning an address to a pointer is done as follows:

- In algorithmics, using the operator: "**@**".
- In the C language, using the operator: "**&**".

Example:

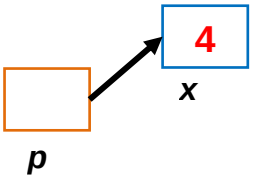
Consider the following code snippet:

`Var x:integer;`	`int x;`
`Var p:^integer;`	`int* p;`
`.....`	`....`
`x ← 4;`	`x=4;`
`p ← @x;`	`p=&x;`



According to the diagram, the pointer **`p`** takes the value 700, which is the address of the integer variable **`x`**. We say that **`p`** ***points to*** **`x`**.

We schematize the pointing operation as follows:



Remarks :

- The type of the pointed variable must match the type of the pointer.

- One of the main sources of errors when working with pointers is the failure to initialize the pointer (assigning no address).
- Initializing a pointer with an arbitrary address is forbidden (for example, the instruction **`p ← 5123;`**).

2.4.1) The Nil value

The value **`Nil`** (**`NULL`** in the C language) is a special value that signifies « no object ». This value can be assigned to any pointer, regardless of its type, to indicate that it does not point to any address.

Example :

The following lines declare a pointer **`p`** and initialize it with the value **`Nil`**. Consequently, **`p`** does not point to any address:

```
Var p:^integer;          int* p;
.....                    ....
p ← Nil;                 p=NULL;
```

2.5) Concept of content

An address is the number of a memory cell, and within this cell resides a value. This value is called the ***content of the pointer*** and should not be confused with its value. The content of a pointer is the value of the memory cell whose address the pointer stores.

However, once a pointer is assigned the address of a variable, it can be used to access the memory cells corresponding to that variable (the value of the variable). This operation is known as ***dereferencing*** or ***indirection***.

Accessing the content of a pointer is done as follows:

- In algorithmics, by using the "**^**" operator after the pointer's name.
- In the C language, by using the "*****" operator before the pointer's name.

Example:

Soit le morceau de code suivant (en algorithmique et en langage C):

Consider the following code snippet (in algorithmics and in C language):

`Var x,y:integer;`	`int x,y;`	Declare two integer variables **`x`** and **`y`**
`Var p:^integer;`	`int* p;`	Declare a pointer to an integer **`p`**
`.....`	`....`	.....
`x ← 4;`	`x=4;`	**`x`** takes the value 4
`p ← @x;`	`p=&x;`	**`p`** points to **`x`**
`y ← p^+1;`	`y=*p+1;`	**`y`** gets 5, which is the content of **`p`** plus 1
`p^ ← 5;`	`*p=5;`	The variable pointed to by **`p`** (**`x`**) gets 5.
`p^ ← ^p*2;`	`*p=*p*2;`	The content of **`p`** (the value of **`x`**) is doubled

Remark :

In the C language, the name of an array is a constant pointer corresponding to the address of the first element of the array.

Example :

If **`T`** is an array, **`*T`** and **`T[0]`** refer to the content of the first element.

2.6) Pointer operations

The only valid operations on pointers are:

2.6.1) Assignment of one pointer to another

Assigning one pointer to another is allowed only if both pointers point to the same type of object (i.e., have the same pointed type).

Example :

```
Var x:integer;
Var p,q:^integer;
......
x ← 5;
p ← @x;
q ← p;
```

After the execution of the above code, **`p`** and **`q`** point to the same memory area (the area reserved for variable **`x`**)

2.6.2) Incrementation and decrementation of a pointer

Incrementing (or decrementing) of a pointer **`p`** by an integer **`n`** signifies that **`p`** now points to the n^{th} object of the pointed type that follows (or precedes) in memory. Incrementing **`p`** by **`n`** is equivalent to increasing the address in **`p`** by **`n`** times the size of the objects. Similarly, decrementing **`p`** by **`n`** is equivalent to decreasing the address in **`p`** by **`n`** times the size of the objects.

Example :

If the pointer **`p`** is of type pointer to integer, the statement **`p ← p+2;`** means increasing the address contained in **`p`** by 2 × 4 bytes, which is equal to 8 bytes.

2.6.3) Comparison

It is possible to compare pointers that point to the same type of objects using the usual comparison operators (**`<`**, **`≤`**, **`>`**, **`≥`**, **`=`**, **`≠`**). These operators allow the comparison of addresses contained in two pointers.

2.6.4) Subtraction of Pointers

Subtracting two pointers is possible, provided that they point to the same type of objects. This difference will provide the number of units of the pointed type placed between the addresses defined by these two pointers.

In other words, the difference between two pointers provides the integer value that needs to be added to the second pointer (in terms of pointer incrementation) to obtain the first pointer.

3) Dynamic Memory Management

In this part of the chapter, we will explore how, during program execution, it is possible to reserve a memory area to store data and then release this area once the processing no longer requires access to it. In this case, we refer to the use of ***dynamic memory allocation***.

3.1) Dynamic Allocation

Dynamic allocation allows the reservation of memory space during the execution of a program. The size of this space is not necessarily known in advance, and the starting address of this space is returned in a pointer.

As a result, access to this dynamically allocated space is done using the pointer, not by using a name, as is the case with static variables.

3.1.1) In algorithmics

Dynamic memory allocation in algorithmics is done using the predefined procedure **`Allocate`** with the following syntax:

```
Allocate(<pointer_name>;
```

This procedure allows the allocation of memory space corresponding to the type pointed to by the pointer provided to the procedure: **`<pointer_name>`**. The address of this space is returned in the pointer **`<pointer_name>`**.

Example:

Consider the following piece of code:

```
Var p:^integer;q:^character
......
Allocate(p);
Allocate(q);
```

This code dynamically reserves two memory spaces (one with a size equal to the size of an integer and the other equal to the size of a character) and puts the starting addresses in the pointers **`p`** and **`q`** respectively.

Access to the reserved spaces is possible for reading, writing, or assignment :

```
Read(p^);
q^ ← 'r';
Write(p^);
```

3.1.2) In C language

In C language, there are two predefined functions for dynamic memory allocation:

a) The `malloc` function

The **`malloc`** function is used to dynamically allocate a certain number of bytes. It takes as input the number of bytes to allocate and returns the address of the first reserved byte. This address is stored in a pointer. The syntax of this function is:

```
<pointer_name> = malloc(<number_of_bytes>);
```

It is then possible to directly assign a value to the dynamically reserved area without the need to identify this area by an identifier.

Example :

```
int* p=malloc(4);
*p=5;
```

Remarks:

- The number of bytes to be reserved is practically related to the type of the value that will be stored in the area. Therefore, the **`<number_of_bytes>`** argument is often provided using the **`sizeof`** function, which returns the size in bytes corresponding to a given type. For example, **`sizeof(int)`** returns 4, which is the number of bytes needed to store an object of type **`int`**.
- The **`malloc`** function also allows the allocation of space for multiple contiguous objects in memory, enabling the dynamic creation of an array. The total number of bytes needed for an array is equal to the number of elements in the array multiplied by the size needed to store one element of the array.

Example:

```
int* t;
t = malloc(10*sizeof(int));
```

This reserves 40 bytes in memory, which can store 10 objects of type **`int`**, and puts the address of the first byte into the pointer **`t`**.

b) The `calloc` function

The **`calloc`** function, like the **`malloc`** function, allocates a memory zone and returns the address of the first byte. However, it additionally initializes all reserved bytes to zero. Its syntax is:

```
<pointer_name> = calloc(<nb_objects>,<object_size>);
```

The **calloc** function appears to be more suitable for dynamically creating arrays.

Example :

```
int* t = calloc(10,sizeof(int));   ⇔   int* t = malloc(10*sizeof(int));
                                       for(i=0;i<10;i++)
                                            t[i]=0;
```

3.2) Memory Deallocation

3.2.1) In algorithmics

La libération de l'espace mémoire pointé par le pointeur **<nom_pointeur>** et qui a été alloué dynamiquement se fait en utilisant la procédure prédéfinie **Libérer**:

Freeing the memory space pointed to by the pointer **<pointer_name>**, which has been dynamically allocated, is done using the predefined procedure **Free**:

```
Free(<pointer_name>);
```

Example:

```
Var p:^integer;
........
Allocate(p);
p^ ← 12;
Free(p);
```

3.2.2) In C language

La libération se fait à l'aide de la procédure prédéfinie **free** comme suit :

The memory release is performed in the C language using the predefined procedure **free** as follows:

```
free(<pointer_name>);
```

When **free** is called, the memory is returned to the system, which can reuse it for other purposes.

Example :

```
int* p;
p=malloc(sizeof(int));
*p=12;
free(p);
```

4) Linked Lists

When we need to store a large amount of information (of the same type), we typically use an array. However, it's not always possible to know the size of the array in advance. In such cases, we need to use a dynamic data structure that can grow or evolve in size. In this structure, it's not necessary to know the number of elements in advance; whenever we need to store information, we allocate the

necessary memory space. These pieces of information, scattered in memory, are linked together using pointers. Such a data structure is called ***a linked list***.

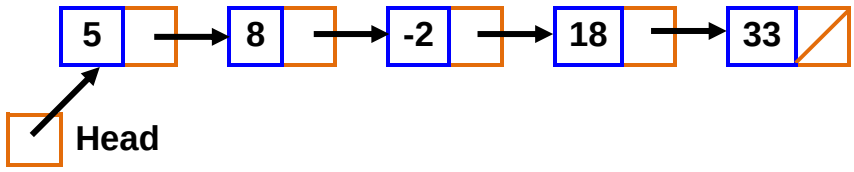
4.1) Definition and presentation

A linked list is a collection of elements (called nodes, cells, …) of the same type, dynamically allocated and linked together using pointers. Each element of the linked list is represented by a record containing the following fields:

- One or more fields containing data of different types (integer, real, array, etc.), as in any structure.
- The last field contains a pointer to the next element.

It is important to note that the list is defined by the address of its first element. This address must be contained in a variable that we will often call the ***head***. Therefore, the head is a pointer that points to the first element.

The following figure presents an example of a linked list formed by 5 elements, each containing an integer field and a pointer field to the next element:



Remarks:

- A simple linked list can only be traversed in one direction (from the head to the tail).
- In practice, elements are created through dynamic allocation.

4.2) Declaration

A linked list is composed of a set of similar elements. Defining the structure of a linked list is equivalent to defining the structure of its elements (one of them).

4.2.1) Declaration in algorithmics

The syntax for declaring a linked list in algorithmic notation is as follows:

```
Type <pointer_type_name> = ^<element_type_name>;
     <element_type_name> = Record
                Begin
                <field1>:<Type1>;
                <field2>:<Type2>;
                ............
                <fieldn>:<Typen>;
                <pointer_field>:<pointer_type_name>;
                End;
Var <head_name>:<pointer_type_name>;
```

The previous syntax first defines a new type called **<pointer_type_name>**, which is a pointer type to the as-yet-undeclared **<element_type_name>**. Then, it declares the type **<element_type_name>**, which is a record containing **n** data fields and a pointer **<pointer_field>** to an element of the same type (in other words, the pointer is of the pre-declared type **<pointer_type_name>**).

Examples:

1) An example of declaring a linked list of integers is as follows:

```
Type List = ^Node;
     Node = Record
               Begin
               val: Integer;
               next: List;
               End;
Var L: List;
```

   In this example, the pointer type is named **List**, and the type describing an element of the list is named **Node**. Each element contains a data field named **val** of type **integer** and a field **next** of type pointer.

2) An example of declaration of a linked list of students:

```
Type StudList = ^Student;
     Student = Record
                Begin
                ID: Integer;
                LastName, FirstName: String[20];
                Marks: Array[9] of Real;
                next: StudList;
                End;
Var L: StudList;
```

4.2.2) Declaration In C language

The syntax for declaring a linked list in the C language is as follows:

```
typedef struct <element_type_name>* <pointer_type_name>;
typedef struct <element_type_name> {
    <Type1> <field1>;
    <Type2> <field2>;
    ............
    <Typen> <fieldn>;
    <pointer_type_name> <pointer_field>;
} <element_type_name>;
<pointer_type_name> <head_name>;
```

Example:

1) The previous linked list of integers is declared in the C language as follows:

```
typedef struct Node* List;
typedef struct Node {
    int val;
    List next;
} Node;
List L;
```

2) The declaration of a linked list of students in C is done as follows:

```
typedef struct Student* StudList;
typedef struct Student {
    int ID;
    char LastName[20], FirstName[20];
    float Marks[9];
    StudList next;
} Student;
StudList L;
```

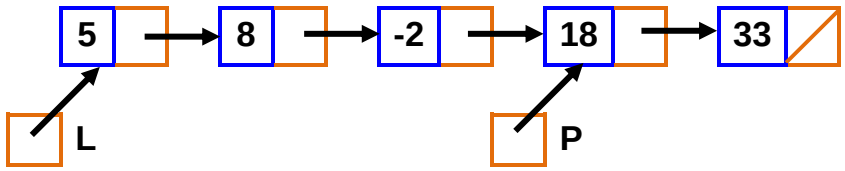
4.3) Accessing fields of a linked list element

Accessing a field of an element pointed to by a pointer is as follows:

- In algorithmics: **<pointer_name>^.<field_name>;**
- In C language: **(*<pointer_name>).<field_name>;**

Example:

Consider the linked list illustrated by the following figure:



Accessing the **val** field of the element pointed to by **p** is done as follows:

In algorithmics: **p^.val**
In C language: **(*p).val**

Remark:

In the C language, the notation **(*<pointer_name>).<field_name>** can be simplified using the structure member pointer operator, denoted as "**->**". Its syntax is:

```
<pointer_name> -> <field_name>
```

Example:

(*p).val is equivalent to **p->val**

5) Operations on Linked Lists

In this section, we will describe some basic operations among those mentioned. Each of these operations can be implemented by a single function or procedure.

For our examples, we will work with lists of integers, using the data structure (the type **`List`**) described in the previous example. The same principles studied are applicable to other data types.

Remarks:

- During creation, be careful to initialize the head of the list to **`Nil`**; otherwise, the chain will not have a final stop, and it will not be possible to locate its end.
- A linked list is known by the address of its first element. If this address is not stored, the list disappears.

5.1) Testing if a list is Empty

We implement this operation with a function **`emptyList`** that takes a linked list as inputand returns **`true`** if it is empty and **`false`** otherwise.

5.1.1) In Algorithmics

```
Function emptyList(L: List): Boolean;
Begin
If L = Nil Then emptyList ← True
Else emptyList ← False
End;
```

5.1.2) In C language

```
int emptyList(List L){
    if(L==NULL)return 1;
    else return 0;
}
```

5.2) Traversing a Linked List

Traversing a linked list has several objectives. It can be done to display its elements, calculate its length (the number of elements), find the maximum or minimum elements, etc.

An example of traversing a linked list is implemented by the following **`displayList`** procedure, which displays the value contained in each element of a linked list whose head is passed as a parameter.

5.2.1) In algorithmics

```
Procedure displayList(L: List);
Var p: List;
Begin
If L = Nil Then Write("The list is empty")
Else  Begin
      p ← L;
      While p ≠ Nil Do
           Begin
           Write(p^.val, " ");
           p ← p^.next;
           End;
     End;
End;
```

5.2.2) In C language

```
void displayList(List L){
  List p;
  if(L == NULL)printf("The list is empty");
  else{
      p = L;
      while(p != NULL){
           printf("%d ",p->val);
           p = p->next;
      }
  }
}
```

5.3) Searching for an element in a linked list

The following function searches if a value **`v`** exists in a linked list having the head **`L`**.

5.3.1) In algorithmics

```
Function search(L: List, v: Integer): Boolean;
Var p: List; found: Boolean;
Begin
p ← L;
found ← False;
While p ≠ Nil and found = False Do
    If p^.val = v Then found ← True
      Else p ← p^.next;
search ← found;
End;
```

5.3.2) In C language

```
int search(List L,int v){
   List p;int found;
   p = L;
   found = 0;
   while(p != NULL && found==0)
      if(p->val == v)found=1;
      else p = p->next;
   return found;
}
```

5.4) Inserting an element

To insert an element into a linked list, three cases are possible:

- Insertion at the beginning (front/ start/head) of the list
- Insertion at the end (tail) of the list
- Insertion in the middle of the list at a fixed position.

In this section, we limit ourselves to the first two cases.

5.4.1) Insertion at the beginning

An example implementation of the insert-at-head function is as follows. This function takes a linked list (identified by its head) and a data value as parameters, adds the data at the beginning of the list, and returns the new head address.

a) In algorithmics

```
Function insertHead(L: List; v: Integer): List;
Var p: List;
Begin
Allocate(p);        // allocate space for the new node
p^.val ← v;         // store the value to be added in the node
p^.next ← L;        // linking
L ← p;              // the head becomes the new node
insertHead ← L;
End;
```

b) In C language

```
List insertHead(List L, int v) {
    List p = malloc(sizeof(node));
    p->val = v;
    p->next = L;
    L = p;
    return L;
}
```

5.4.2) Insertion at the end

An example implementation of the insertion-at-tail function is as follows:

a) In algorithmics

```
Function insertTail(L: List, v: Integer): List;
Var p, q: List;
Begin
Allocate(p);          //allocate space for the new node
p^.val ← v;           //put the value to be added in the node
p^.next ← Nil;        //the new element becomes the last one
If L = Nil Then       //special case when the list is empty
    L ← p             //the new element becomes the head
Else Begin
     q ← L;           //the auxiliary pointer starts from the first element
     While q^.next ≠ Nil Do    //search for the last element
               q ← q^.next;
     q^.next ← p;     //linking
     End;
insertTail ← L;
End;
```

b) In C language

```
List insertTail(List L, int v) {
    List p, q;
    p = malloc(sizeof(node));
    p->val = v;
    p->next = NULL;
    if (L == NULL) L = p;
    else {
        q = L;
        while (q->next != NULL)
            q = q->next;
        q->next = p;
    }
    return L;
}
```

5.5) Deleting an element

There are also three possible cases:

- Delete the first element of the list.
- Delete an element in the middle of the list.
- Delete the last element of the list.

Note that deletion should be done carefully; deleting an element without precaution will prevent access to subsequent elements.

5.5.1) Deletion at the beginning

An example implementation of the deletion-at-head function is as follows:

a) In algorithmics

```
Function deleteHead(L: List): List;
Variable p: List;
Begin
If L ≠ Nil then      //if the list is not empty
   Begin
   p ← L;            //store the address of the head in pointer p
   L ← L^.next;      //move the head to the next element
   Free(p);          //free the space occupied by the first element
   End;
deleteHead ← L;
End;
```

b) In C language

```
Liste deleteHead(List L){
  List p;
   if(L != NULL){
      p = L;
      L = L->next;
      free(p);
   }
   return L;
}
```

5.5.2) Deletion at the end

a) In algorithmics

```
Function deleteTail(L: List): List;
Var p, q: List
Begin
If L ≠ Nil Then      //If the list is not empty
   Begin
   If L^.next = Nil Then   //If the list contains only one element
      Begin
      Free(L);   //Free the space occupied by the head
      L ← Nil;   //The head points to Nil
      End
```

```
      Else Begin   //If the list contains at least 2 elements
           p ← L;
           While p^.next ≠ Nil Do //While we are not at the last element
                  Begin
                  q ← p;          //Keep the previous element
                  p ← p->next;    //Move p to the next
                End;
//At the end of the loop, p points to the last element, and q to the second-to-last.
           q^.next ← Nil; //Indicate that the second-to-last becomes the last
           Free(p);       //Free the memory occupied by the last element
           End;
      End;
   deleteTail ← L;
   End;
```

b) In C language

```
   List deleteTail(List L){
     List p,q;
     if(L != NULL){
         if(L->next == NULL){
              free(L);
              L = NULL;
         }
         else{
              p = L;
              while(p->next != NULL){
                   q = p;
                   p = p->next;
              }
              q->next = NULL;
              free(p);
         }
     }
     return L;
   }
```

5.5.3) Destruction of a linked list

The following function allows the destruction of a linked list by successively deleting the first element as long as the list is not empty.

a) In algorithmics

```
Function destroyList(L: List): List;
Var p: List;
Begin
While L ≠ Nil Do     //repeat until the list is empty
    Begin
    p ← L;           //store the address of the head in the pointer p
    L ← L^.next;     //move the head to the next element
    Free(p);         //free the space occupied by the first element
    End;
destroyList ← L;
End;
```

b) En langage C

```
Liste destroyList(List L){
   List p;
   while(L != NULL){
        p = L;
        L = L->next;
        free(p);
   }
   return L;
}
```

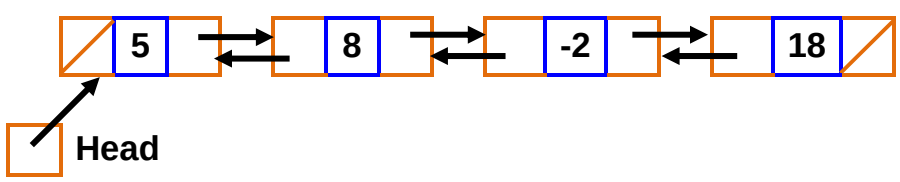
6) Doubly Linked Lists

6.1) Presentation

There are also other variants of linked lists, called ***doubly linked lists*** or ***bidirectional lists***, which can be traversed in both directions, from the first node to the last node, and vice versa. This requires adding another pointer field to the node structure, called **`prev`** which should contain the address of the previous node.

As a result, the elements of a doubly linked list are linked by two pointers: one pointer to the next element and one pointer to the previous element. The pointer to the previous element of the first element is set to **`Nil`** because it has no predecessor.

An example of a doubly linked list with 4 elements is illustrated in the following figure:



6.2) Declaration

An example of declaring the doubly linked list illustrated in the previous figure is as follows:

In algorithmics

```
Type List =^node;
   Node = Record
          Begin
          val:Integer;
          next,prev:List;
          End;
Var L:List;
```

In C language

```
typedef struct Node* List;
typedef struct Node{
            int val;
            List next,prev;
  }Node;
List L;
```

Remark :

To efficiently exploit this type of list, it is preferable to use two pointers, one for the head of the list and another for the tail. This way, we can, for example, display the list in reverse order by traversing the list from the end and using the **`prev`** fields.

6.3) Operations on Doubly Linked Lists

All operations possible on singly linked lists are also applicable to doubly linked lists. Examples include insertion at the head, insertion at the tail, search, display, deletion, etc.

The only difference is that we need to manage two linkages (the **`suiv`** and **`prev`** pointers) and handle two heads (beginning and end) if our list is defined by two pointers: head and tail.

To illustrate these operations, we will use the example of inserting a new element at the head of a doubly linked list.

```
Function insertHead(L:List, v:integer):List;
Var p:List;
Begin
Allocate(p);
p^.val ← v;
p.prev ← Nil;
p^.next ← L;
If L ≠ Nil then      // If it is not the first insertion in the list.
     L^.prev ← p;
L ← p;
insertHead ← L;
End;
```