People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
Mohamed El Bachir El Ibrahimi University – Bordj Bou Arréridj
Faculty of Mathematics and Computer Science
Department of Computer Science

Course

Algorithmic and Data Structures 2

With Solved exercises in algorithmic language and C language

Dr. SAIFI Lynda

2024/2025

Foreword

This course is intended for first-year students in Mathematics and Computer Science, and Computer science engineering with the aim of providing them with a reference document that is both general and concise.

This document contains, for all the functionalities and concepts described in the canvas of this course, simple definitions and examples in algorithmic formalism and in the C language in order to facilitate both understanding and practice for the students.

Prerequisites

It is recommended to have knowledge of:

- Basic mathematical and logical concepts
- Basic computer functions
- Concepts learned in the ASD1 course

Targeted Skills

Upon completion of this course, the student will be able to:

- Analyze a real problem and potentially break it down into sub-problems.
- Understand the difference between the role of a programmer and that of a user.
- Construct an arithmetic or logical expression using algorithmic formalism.
- Write the solution to an analyzed problem in the form of an algorithm.
- Execute the algorithm and identify its shortcomings and bugs.
- Master the required data structures.

Summary

Foreword

Prerequisites

Targeted skills

Part 1: Courses

Chapter 1: Functions and Procedures

1. Introduction
2. Procedures
   2.1. Definition
   2.2. Syntax
   2.3. Procedure call
   2.4. Examples
   2.5. Examples in C language
   2.6. Parameter passing
      2.6.1. Pass by value
      2.6.2. Pass by address
   2.7. Remarks
3. Functions
   3.1. Definition
   3.2. Syntax
   3.3. Function call
   3.4. Remarks
   3.5. Examples
   3.6. Examples in C language
   3.7. Local Variables and Global Variables
4. Recursion
   4.1. Definition
   4.2. Examples
   4.3. Examples in C language

Chapter 2: Files

1. Introduction
2. Definition
3. Types of files
4. File manipulation
   4.1. Open
   4.2. Read
   4.3. Write
   4.4. Close

5. Remakes
6. Examples
7. Examples in C language

Chapter 3: Linked lists

1. Introduction
2. Pointer
   2.1. Syntax
   2.2. Examples
   2.3. Examples in C language
3. Dynamic memory management
   3.1. Creating a dynamic variable of type t
   3.2. Assignment between pointers
   3.3. Using the element pointed to by a pointer
   3.4. Freeing up space occupied by a dynamic variable
   3.5. Assigning the Nil value to a pointer
   3.6. Pointer to an Array
   3.7. Examples
   3.8. Examples in C language
4. Linked lists
   4.1. Definition
   4.2. Syntax
   4.3. Examples
   4.4. Examples in C language
   4.5. Comparison between Arrays and Lists
5. Operations on Linked Lists
   5.1. Creating a List
   5.2. Deleting an Element
   5.3. Traversing a List to Display Elements
   5.4. Searching for a Value in a List
6. Doubly Linked Lists
   6.1. Definition
   6.2. Syntax
   6.3. Examples
   6.4. Examples in C language
7. Special Linked Lists
   7.1. Stacks
      7.1.1. Definition
      7.1.2. Remarks
      7.1.3. Main operations
   7.2. Queues
      7.2.1. Definition
      7.2.2. Remarks
      7.2.3. Main operations

Part 2: Solved exercises

Tutorial 1: Subprograms
Solution
Tutorial 2: Linked lists
Solution

Bibliographic references

Chapter 1 Subprograms

1. Introduction

When faced with a complex problem, we often need a lengthy or complicated algorithm. In such cases, the strategy is to divide and conquer, breaking down the problem into sub-problems. Subsequently, an algorithm is written for each sub-problem, referred to as a "module" or "subprogram" or "subroutines". Finally, we integrate everything by calling these modules in the main algorithm.

2. Procedures

2.1. Definition

A procedure is a subprogram that performs one or more actions and can return 0, 1, or multiple results. It’s a part of algorithm.

2.2. Syntax

Procedure procedure-name (list of formal parameters with their types separated by ',')
Declaration of local variables of the procedure
Begin
Body of the procedure
End

2.3. Procedure call

The procedure call is the instruction that allows its use in the main algorithm by executing the code with effective parameters.

Procedure-name (list of real or effective parameters)

2.4. Example

A procedure that swaps the content of two integer variables, x and y, passed as parameters.

Procedure swap (var a: integer, var b: integer)
Variable tmp: integer
Begin
tmp ← a;
a ← b;
b ← tmp;
End

Algorithm swap_main
Variables x, y: integer
Begin
Write('Enter the value of x:')
Read(x)
Write('Enter the value of y:')

```
Read(y)
Write('before procedure call x= ',x,' and y= ',y)
Swap(x, y)
Write('after procedure call x= ',x,' and y= ',y)
End
```

2.5. Example in C language

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int x, y;
    printf("Enter the value of x: ");
    scanf("%d", &x);

    printf("Enter the value of y: ");
    scanf("%d", &y);
    printf("Before procedure call x = %d and y = %d\n", x, y);
    swap(&x, &y);
    printf("After procedure call x = %d and y = %d\n", x, y);
    return 0;
}
```

2.6. Parameter Passing

2.6.1. Pass by value

Formal parameters represent local variables within the subprogram, and any changes to the parameter have no effect on the effective parameter. Values of real parameters are retrieved in formal parameters for processing. This passage concerns the data parameters or **input variables**. They are passed by value.

2.6.2. Pass by address

Formal parameters are associated with the addresses of real parameters. Changes to formal parameters affect effective parameters. This mode applies to **output variables**. These parameters are passed by address.

2.7. Remarks

- The first line of the procedure is called the "header" or "signature" and "prototype" in C language.

- The parameters passed by address are preceded by the keyword “var” or the the keyword “Output” or the keyword “I/O”. In C language, they are preceded by a star “*”.

- When calling the procedure, the real variables replace the formal variables, they must match in type, number, and order too.

- The formal parameters and the effective parameters can have the same name.

- In C language, the effective parameters are called "arguments".

3. Functions

3.1. Definition

A function is a subprogram that must return a single value of scalar type (integer, real, character, boolean). This value is stored in a variable with the same name as the function. The function is a particular case of the procedure (the procedure is the general case of subroutines).

3.2. Syntax

Function function-name (list of formal parameters with their types separated by ',') : result_type
Declaration of local variables of the function
Begin
List of actions of the function
End

3.3. Function call

Since a function returns a single result, it is called within the main algorithm by assigning the returned value to a variable declared in that algorithm.

The call takes the following form:

Identifier ← function-name (list of effective parameters)

3.4. Remarks

- The types of the identified variable and the returned result must be compatible.
- The formal parameters of the function can only be data parameters (passed by value) since they are input variables. Unlike procedures that can contain data parameters and result parameters (passed by address), they are the output variables.

- When calling the function or procedure, the effective or real parameter can be: a variable, an arithmetic or logical expression or a call to another function that returns a value of the same type as the corresponding formal parameter.

- A main program can call several functions, and each function can be called several times by the same main program.

3.5. Example

Write a function that takes two integers, a and b, as parameters and calculates a^b.

Function **power** (a: integer, b: integer): integer
Variable p, i: integer
Begin
p ← 1
For i going from 1 to b do p ← p * a
End for
power ← p
End
Algorithm power_main
Var x, y, z: integer
Begin
Write('Enter the value of x:')
Read(x)
Write('Enter the value of y:')
Read(y) z ← power(x, y)
Write(x, ' to the power of ', y, ' is ', z)
End

3.6. Example in C language

```
#include <stdio.h>
int x, y, z;
int power(int a, int b);
// Main function
int main() {
printf("Enter the value of x: ");
scanf("%d", &x);
printf("Enter the value of y: ");
scanf("%d", &y);
z = power(x, y);
printf("%d to the power of %d is %d\n", x, y, z);
return 0;
}
// Function to calculate power
int power(int a, int b) {
int p = 1;
for (int i = 1; i <= b; i++)
p *= a;
return p;
}
```

3.7.Local Variables and Global Variables

Formal or real parameters are global variables known throughout the main program and the subprogram. However, local variables are declared within a subprogram and are not known in the main program.

For example, in the swap procedure, variables a and b are "global," and tmp is "local." In the power function, variables a and b are "global," and variables p and i are "local."

4. Recursion

4.1.Definition

A recursive function is a function that calls itself. It is defined by at least one base case and at least one general case.

In the base case, we describe the cases for which the result of the function is easy to calculate: the value returned by the function is directly defined.

On the other hand, in the general case: the function is called recursively and the result returned is calculated using the result of the recursive call. At each recursive call, the value of at least one of the effective parameters of the function must change. It is always necessary to ensure that each general case converges to a base case.

4.2.Examples

Example 1

Factorial of an integer.

Function Fact (n: integer): integer
Variable
Begin
 If (n = 0) THEN Fact ← 1
Else Fact ← n * Fact(n-1)
End

Example 2 In C language

A recursive subprogram that calculates the sum of the first N numbers.

```
int sumN(int n) { if (n == 1) return 1; else return (n + sumN(n-1)); }
```

Example 3

A recursive subprogram that calculates the sum of the first N squares.

For example, if N is 3, this subprogram will calculate $1^2 + 2^2 + 3^2$. This subprogram is defined only for N greater than 0.

Function SumFirstSquares (N: integer): integer
VAR f: integer

Begin f ← 1
IF (N = 1) THEN SumFirstSquares ← f
ELSE SumFirstSquares ← (N*N) + SumFirstSquares(N-1)
End.

Chapter 2 Files

1. Introduction

Files are essential in algorithmic processes, as they allow data to be stored and manipulated persistently. Unlike variables, which are temporary and disappear once the program is executed, a file remains on the hard drive, enabling information to be preserved across multiple executions of the program. Manipulating files involves reading, writing, and modifying data within these files.

2. Definition

A file is an organized collection of data stored on an external medium (such as a hard drive). Indeed, manipulating a file means reading or writing data in this file. Some of the reasons to use files are:

- To save data permanently.
- To retrieve information without needing to re-enter it.
- To share data between different programs.

3. Types of Files

The types of files are:

- **Text files**: Contain readable data (e.g., .txt).
- **Binary files**: Contain data that is not directly readable (faster and more compact).

4. File Manipulation

Here are the main operations that can be performed on a file:

4.1.Open: This operation means connecting to the file (for reading/writing).

Syntax

Open (file, mode), where the mode can be:

- **read**: to read the file
- **write**: to create/write to the file
- **append**: to add to the end of the file

4.2. Read: This operation allows extracting data from the file.

Syntax

Read (file, variable)

4.3. Write: This operation allows inserting data into the file.

Syntax

Write (file data)

4.4. Close: This operation frees the file once done.

Syntax

Close (file)

5. Remarks

- Always open a file before reading or writing.
- Close the file after finishing.
- Check that the file exists before reading.
- When reading, use a loop to read until the end of the file.

6. Example

Write an algorithm that asks the user to enter the names of 5 students, stores these names in a file called "eleves.txt", and then reads the file and displays the students' names on the screen.

```
Algorithm FileManipulation
Variables:
   file : TextFile
   name : String
   i : Integer
Begin
   // 1. Writing to the file
   Open (TextFile, "write")
   For i from 1 to 5 do
      Display ("Enter the name of student ", i, " : ")
      Read (name)
      Write (TextFile, name)
   EndFor
   Close (TextFile)
   // 2. Reading the file
   Open (TextFile, "read")
   Display ("List of students:")
   While Not EndOfFile (TextFile) do
      Read (TextFile, name)
      Display (name)
   EndWhile
   Close (TextFile)
End
```

7. Example in C language

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
   FILE *file;
   char name[100];
   int i;
   // 1. Writing to the file
   file = fopen ("eleves.txt", "w");  // Open the file in write mode
   if (file == NULL) {
      printf("Error opening the file for writing.\n");
      return 1; // Exit the program in case of an error
   }
   // Ask for student names
   for (i = 1; i <= 5; i++) {
      printf ("Enter the name of student %d: ", i);
      fgets (name, 100, stdin); // Read the student's name
      fprintf (file, "%s", name); // Write the name to the file
   }
   fclose (file);  // Close the file after writing
   // 2. Reading the file
   file = fopen ("eleves.txt", "r");  // Open the file in read mode
   if (file == NULL) {
      printf ("Error opening the file for reading.\n");
      return 1; // Exit the program in case of an error }
   // Display student names
   printf("\n List of students: \n");
   while (fgets(name, 100, file) != NULL) {
      printf("%s", name); // Display each name }
   fclose(file);  // Close the file after reading
   return 0;}
```

Chapter 3 Linked List

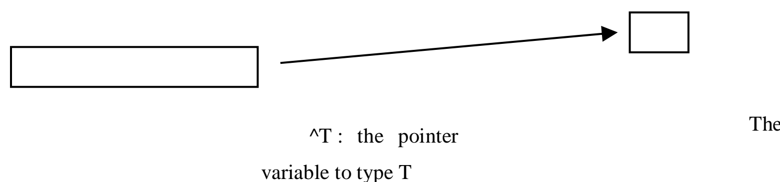
1. Introduction

Static variables are variables declared before the execution of programs like the simple or compound type variables seen previously. These static variables have major drawbacks; the memory space reserved for the variable is either too small or too large. In the first case, there is a risk of running out of space, for example, when the number of elements in an array is insufficient. In the second case, we face memory wastage.

The dynamic variables we will see in this chapter address this issue. They allow for the reservation or release of memory space as needed during the execution of the program.

2. Pointers

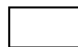
Let T be any variable of type T, and let ^T be the pointer type that points to type T. The pointer variable to T (^T) contains, in the form of a memory address, the access mode to the content of the variable of type T. This variable does not have an identifier but is accessible through its pointer.



The pointer that contains the address occupies a 4-byte space in the main memory, and the variable of type t usually occupies 1 byte. In the C language, a variable of type char occupies 1 byte, int occupies 2 bytes, and float occupies 4 bytes. Regarding arrays, the pointer to an array stores the address of the first element of the array. Additionally, it is possible to point to a pointer (store the address of a pointer) by using two symbols ^^ (in C language: **), whether for declaration or for accessing the information.

2.1. Syntax

Define a type t.
Declare ptr: pointer_to_t or ^t.



2.2. Example

```
p1 : ^ int,  x :int
p2,p3 : ^ real
p4 : ^ char
p5 : ^^ int
```

2.3. Example in C language

```
int   * p1 , x;
float * p2 , * p3;
char  * p4 ;
```

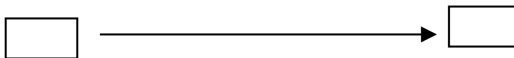
int * * p5;

3. Dynamic Memory Management

Allocation and de-allocation of a memory area according to the need offer the possibility of dynamic memory management. After declaring a pointer to t, we can perform the following operations on and through the pointer ^t:

3.1. Creating a dynamic variable of type t

The "allocate (ptr)" instruction reserves a memory area of type t and of sufficient size for this type, generally 1 byte, and then, his address in stoked in the variable ptr. This area will be accessible through the identifier ptr^.



Example

Allocate (p2)
Allocate(p4)

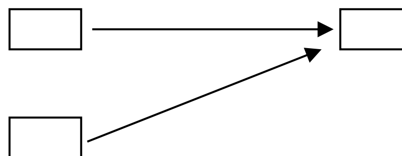
Example in language

p1 =&x ; /* assignment of the address of the variable x of type int to the pointer p1. */
malloc (p2) ;
malloc (p4) ;

3.2. Assignment between pointers

Pointers are particular variables; they contain memory addresses. When assigning one pointer to another, it means they contain the same address, and thus they point to the same area. A condition for assignment, when declaring these pointers, is that they must point to the same type of data.

ptr1, ptr2: pointer to t
allocate (ptr1)
ptr2 ← ptr1



Example

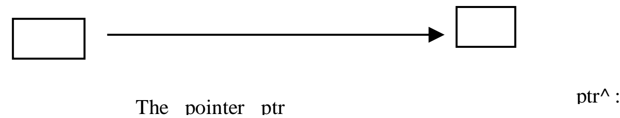
p3 ← p2
p5^ ← p1

Example in C language

p3 = p2 ;
p5* = p1 ; /* pointer that points to another pointer*/

3.3. Using the element pointed to by a pointer

Once the pointer ptr is declared and allocated, it is possible to use the pointed element (the variable of type t). This is done by the identifier ptr^ (^ is called the dereference operator as it allows moving from the pointer to the pointed element).



When accessing the pointed variable ptr^, the same instructions as for a simple variable can be applied, such as reading, writing, and assigning an expression of type t:

Read (ptr^),
Write (ptr^),
ptr^ <- expression of type t

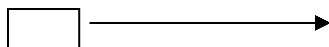
Example
p3^← 12.6
Read (p2^),
Write (p3^), 12.6
p4^ ← ‘g’
P5^^ ← 45

Example in C language

```
*p3 = 12.6;
scanf("%f", p2);
printf("%f\n", *p3);
*p4 = 'g';
**p5 = 45;
```

3.4. Freeing up space occupied by a dynamic variable

The "deallocate (ptr)" instruction frees the memory area of type t reserved previously and possibly used. This area will be available for allocation by other variables. However, this instruction does not erase the memory address stored in the pointer, and if this deallocated pointer is called upon, it returns information that does not make sense.



Example
Deallocate (p2)
Deallocate (p4à

Example in C language
Free (p2);
Free (p4);

3.5. Assigning the Nil value to a pointer

To "empty" a pointer, i.e. to cancel the address it contains, we assign it a predefined value named Nil (or Null). However, putting Nil in a pointer does not free up the location it pointed to. The location becomes unrecoverable because the link to this location has been cut by the Nil value. De-allocation must be done before assigning the pointer with Nil.

ptr ← Nil

Good programming rule: as soon as a pointer is de-allocated, it should be assigned the value Nil so that it does not retain a memory address that no longer physically exists.

Example
p1 ← Nil
p2 ← Nil

Example in C language
p1 = Null;
p2 = Null;

Example 1 (General example to Understand)

Algorithm: Pointers

Variables
P1, P2: ^ integer
ch: ^ string (20)

Begin
Allocate (ch)

ch^ <- "Mohammed"

Write (Ch^)

Allocate (P1)

Read (P1^)

Write (P1^)

P2 <- P1

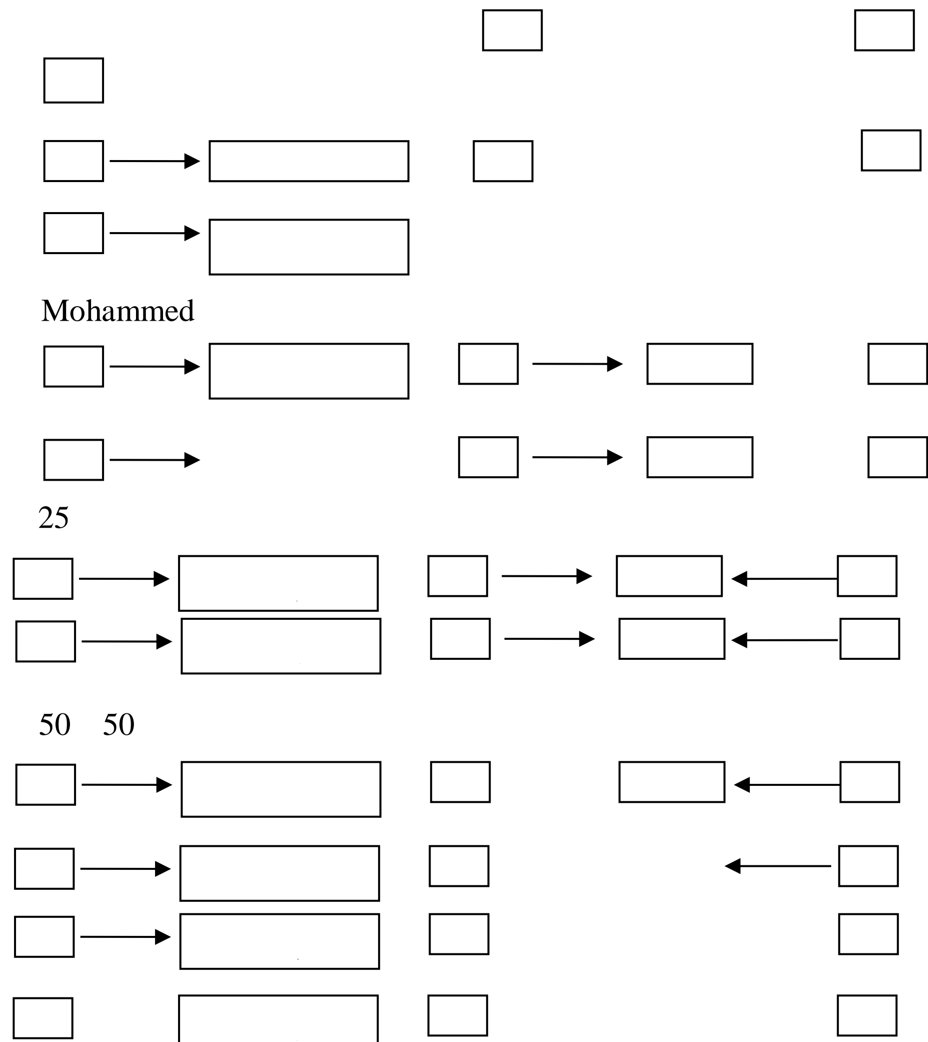
P2^ <- P1^ + P2^

Write (P1^, P2^)

P1 <- Nil

Deallocate (P2)

P2 <- Nil



```
ch <- Nil
```

End

What is the mistake?

Example 2: Exchange the value of two real pointers

```
Algorithm: SwapPointerValues
Variables: temp: real
P1, P2: ^ real
Begin
Allocate (P1, P2)
Write ("Please provide two real numbers")
Read (P1^, P2^)
temp <- P1^
P1^ <- P2^
P2^ <- temp
Write (P1^, P2^)
End
```

Solution 2: Only exchange the addresses of the two pointers

```
Algorithm: SwapPointerAddresses
Variables:
P1, P2, temp: ^ real
Begin
Allocate (P1, P2)
Write ("Please provide two real numbers")
Read (P1^, P2^)
temp <- P1
P1 <- P2
P2 <- temp
Write (P1^, P2^)
End
```

Example 2 in C language

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
   double *p1, *p2, temp;

   // Allocate two pointers to real numbers
   p1 = (double *) malloc(sizeof(double));
   p2 = (double *) malloc(sizeof(double));
```

```
// Read two real numbers
Printf ("Please enter two real numbers: ");
Scanf ("%lf %lf", p1, p2);
// Swap the values pointed to by p1 and p2
temp = *p1;
*p1  = *p2;
*p2  = temp;

// Display the first value (which was initially the second)
printf("After swapping, first value = %lf\n", *p1);

// Free the memory and set the pointers to NULL
free(p1);
free(p2);
p1 = NULL;
p2 = NULL;
return 0;}
```

Example 3: The sum of two integer pointers

```
Algorithm: SumPointers
Variables: s: integer
P1, P2: ^ integer
Begin
Allocate (P1, P2)
Write ("Please provide two integer numbers")
Read (P1^, P2^)
s <- P1^ + P2^
Write (s)
End
```

3.6. Pointer to an Array

The pointer to an array in the C language provides additional precision. There is the standard pointer, that is, a pointer to an integer, commonly used to iterate through an array. It is declared as follows: int *P;

This is a pointer to the first element of the array (it holds the address of the first element). Access to the following elements is done by incrementing the address according to the type of the array elements, as follows: *(P + i)

Using the indirection operator * and starting from address P. Note that P + 0 = P because the addresses of array elements are consecutive thanks to pointer arithmetic, as explained in Example 4.

The second type is a pointer to an entire fixed-size array, which points to a data structure of type "array" .Here is the declaration of a pointer to an array of three elements: int (*P)[3];
The statement (*P)[i] is used to access the elements of the array, as explained in Example 5.

Example 4

```
#include <stdio.h>
int main() {
   int arr[5] = {10, 20, 30, 40, 50}; // An array of 5 integers
   int *ptr = arr;
   // A pointer to the first element of the array; arr is equivalent to &arr[0]
   // Displaying the array elements using the pointer
   for (int i = 0; i < 5; i++) {
      printf("Element %d: %d\n", i, *(ptr + i)); // address incrementing
      // *(ptr + i) accesses the elements of the array
   }
   return 0;
}
```

Example 5

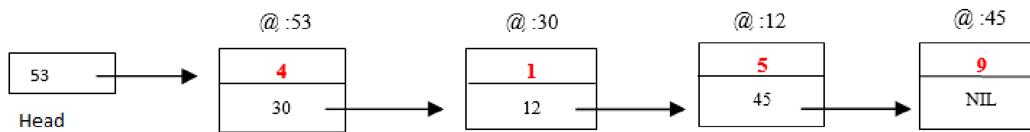
```
#include <stdio.h>
int main() {
   int arr[5] = {1, 2, 3, 4, 5};

   // Declaration of a pointer to an array of 5 integers
   int (*ptr)[5] = &arr; // &arr gives the address of the entire array
   // Accessing the elements via the pointer to the array
   for (int i = 0; i < 5; i++) {
      printf("Element %d: %d\n", i, (*ptr)[i]);
   }
   return 0;
}
```

4. Linked lists

4.1. Definition

A linked list is a collection of elements or nodes that are connected. These elements are of the same type, and each has a memory address. Linked lists use dynamic memory management, also known as linear programming, which involves the use of pointers.
The elements of a list, or nodes, are of structure (record) type. This structure is composed of two parts: the first part of the node contains the information or data of any simple or composite type, and the second part of the structure is a pointer that holds the address or access to the next node in the list.

Example: A linked list containing the four integer values: 4, 1, 5, 9.



This list is composed of four nodes, each node containing an integer data and a pointer that points to the next node (storing the address of the next node). The address of the first node is stored in a pointer usually called "head". When "head" is equal to Nil, it means that the list is empty. The pointer of the last element in the list contains the value Nil because there is no subsequent node.
In so-called **"circular linked lists",** the pointer of the last element points back to the first element.

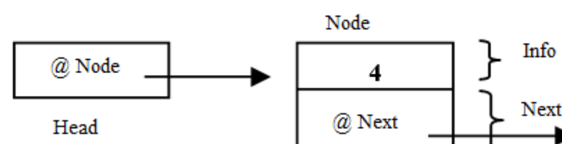
4.2. **Syntax**:

To create a linked list, you must start by defining the nodes: this is a record type that contains the information and the pointer. The list type is a pointer to a node. It can be named, for example, "List" or simply left as "^Node". To manage the linked list, two variables of type ^Node need to be created. The first variable, for example called "Head", contains the head of the list — meaning the address of the first element. The second variable, for example named "P", is a pointer used to create the elements of the list (the nodes). Then, each node is added to the list as follows. You can notice in this syntax "the principle of recursion", where each node uses a pointer variable to another node.

```
Type List = ^Node
Type Node = Record
   Info: simple or composite type
   Next: List
End Record
Variables Head, P: List
```

4.3. **Example**

```
Type Node = Record
   Info: integer
   Next: ^Node
End Record
```




```
Variables Head: ^Node
```

4.4. **Example in C language**

```
typedef struct Node {
   int info;            // Information: integer type
   struct Node* next;      // Pointer to the next node
```

} Node;
typedef Node* List;   // List is a pointer to a Node
// Declaration of variables
List head, p;

4.5. Comparison between Arrays and Lists

Structure	Type of Structure	Storage	Size	Position of Information	Access to Information	Insertion / Deletion
Array	Simple, Static	Contiguous (successive in memory)	Fixed at declaration	By its index	Directly by index	Slow except at the end (requires shifting)
List	Structured, Dynamic	Non-contiguous	Dynamic (easy to grow/shrink)	By its address	Sequentially via each element's pointer	Fast (if the previous pointer is known)

5. Operations on Linked Lists

5.1. Creating a List

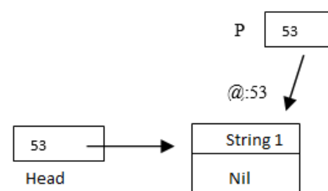
In this section, we start by creating a linked list with 2 elements of type string. To do this, the algorithm prepares the first element, inserts it at the head, then adds the second element and inserts it at the head as well: the first created element becomes the “next” of the newly inserted one. The following algorithm allows creating a list of N elements.

Type declarations for List and Node

Type List = ^ Node
Type Node = Structure
  Info: String
  Next: List
End Structure

Algorithm creation-List

Head, P: List
BEGIN
  Head ← Nil
  Allocate (P)
  Read (P^.Info)
  P^.Next ← Nil

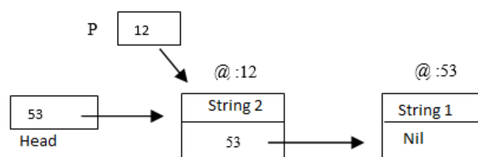


  Head ← P   /* The Head pointer points to P: we have inserted the first allocated element at the head */

  Allocate (P)   /* Now we need to prepare the second element */
  Read (P^.Info)

  P^.Next ← Head   /* The second element is inserted at the head of the list, and the first created element becomes the next of the inserted one */

```
   Head ← P
END
```



Example in C language

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
   char info[100]; // Taille à ajuster selon besoin
   struct Node* next;
} Node;

typedef Node* List;

void CreateList(List* head) {
   List p;
   *head = NULL;
   // Allocate and insert first element
   p = (List)malloc(sizeof(Node));
   if (p == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
   printf("Enter info for first element: ");
   scanf("%s", p->info);
   p->next = NULL;
   *head = p;
   // Allocate and insert second element
   p = (List)malloc(sizeof(Node));
   if (p == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
   printf("Enter info for second element: ");
   scanf("%s", p->info);
   p->next = *head;
   *head = p;
}
```

5.2. Deleting an Element

This example shows how to delete the first element of a list using a procedure, where the head pointer is passed as a parameter.

Procedure DeleteFirstElement (Var Head: List)

```
Variables:

P: List

BEGIN
   IF Head ≠ Nil THEN   /* The list is not empty */
      P ← Head         /* P points to the first element of the list */
      Head ← P^.Next   /* The head of the list points to the second element */
      Deallocate(P)
   ELSE
      Write("The list is empty")
   ENDIF
END
```

Example in C language

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
   char info[100];
   struct Node* next;
} Node;

typedef Node* List;

void DeleteFirstElement(List* head) {
   List p;

   if (*head != NULL) { // The list is not empty
      p = *head;        // p points to the first element
      *head = p->next;  // head now points to the second element
      free(p);          // deallocate the first element
   } else {
      printf("The list is empty.\n");
   }
}
```

5.3. Traversing a List to Display Elements

This example shows how to traverse the entire list in order to display its elements.

Procedure TraverseList (P: List)

```
BEGIN
   P ← Head      /* P points to the first element of the list */
   WHILE P ≠ Nil DO      /* If the list is empty, Head is Nil */
      Write (P^.Info)  /* Display the value stored at the address pointed by P */
      P ← P^.Next      /* Move to the next element */
   END WHILE
END
```

Example in C language

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
   char info[100];
   struct Node* next;
} Node;

typedef Node* List;

void TraverseList(List head) {
   List p = head; // P points to the first element

   while (p != NULL) { // While the list is not empty
      printf("%s\n", p->info); // Display the info
      p = p->next; // Move to the next element
   }
}
```

5.4. Searching for a Value in a List

A procedure that searches for a value (provided as a parameter) in the list passed as a parameter; assuming the list's elements are of type integer.

Procedure SearchValue (Head: List, Value: int)

```
Variables:
P: List
Found: boolean
BEGIN
   IF Head ≠ Nil THEN /* The list is not empty */
      P ← Head
      Found ← False
      WHILE P ≠ Nil AND NOT Found DO
```

```
        IF P^.Info = Value THEN /* The searched element is the current element */
          Found ← True
        ELSE
          P ← P^.Next /* Move to the next element in the list */
        ENDIF
      END WHILE
      IF Found THEN
        Write("The value ", Value, " is in the list")
      ELSE
        Write("The value ", Value, " is not in the list")
      ENDIF
    ELSE
      Write("The list is empty")
    ENDIF
END
```

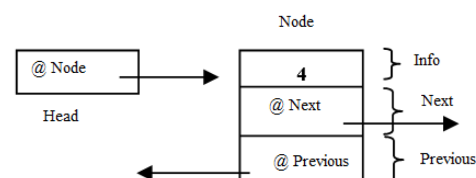
6. Doubly Linked Lists

6.1. Definition

We previously saw simple linked lists, where each element of the list contains a pointer pointing to only one subsequent element. In contrast, in a doubly linked list, each element points to two elements. In this type of list, each node is a structure containing three elements: the information or data and two pointers. The first pointer points to the previous element in the list and is generally called “Previous”; the second pointer, typically called “Next”, points to the next element in the list, as seen earlier in simple linked lists. The type of the “info” variable can be either a simple or composite type, depending on the needs. The “Previous” pointer of the first element of the list and the “Next” pointer of the last element contain the value Nil. To create this type of list, you need to start by declaring two variables, Head and Tail, of type ListDC. When initializing a doubly linked list, the Head and Tail pointers contain the value Nil (since the list is still empty).

6.2. Syntax

```
Type ListDC = ^Node
Type Node = Record
  Previous: ListDC
  Info: simple or composite type
  Next: ListDC
End Record
```

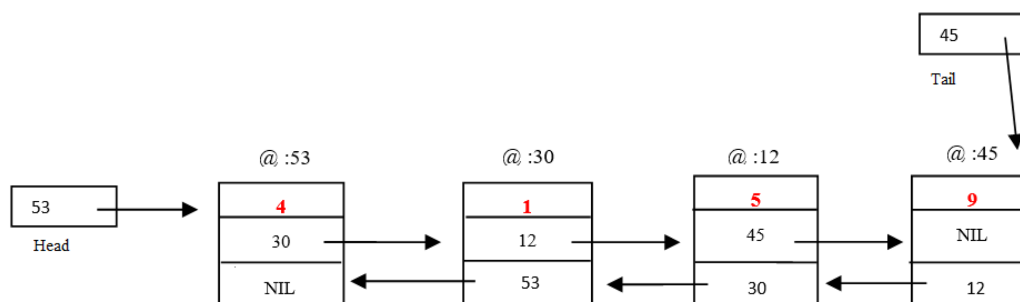


6.3. Example

```
Type ListDC = ^Node
Type Node = Record
  Previous: ListDC
```

Info: Integer
Next: ListDC
End Record

Variables
P, Head, Tail: ListDC



6.4. Example in C Language

```
#include <stdio.h>
#include <stdlib.h>

// Definition of the doubly linked list node
typedef struct Node {
    int info;                // Information: integer type
    struct Node* previous;   // Pointer to the previous node
    struct Node* next;       // Pointer to the next node
} Node;

typedef Node* ListDC;        // ListDC is a pointer to a Node

// Function to add a new node at the end of the doubly linked list
void addNode(ListDC* head, ListDC* tail, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->info = value;
    newNode->next = NULL;
    newNode->previous = *tail;  // Set the previous pointer to the tail

    if (*tail != NULL) {
        (*tail)->next = newNode;  // Set the next pointer of the old tail to the new node
```

```
    }
    *tail = newNode;            // Update the tail to the new node

    if (*head == NULL) {
        *head = newNode;       // If the list was empty, the new node becomes the head
    }}

// Function to display the list from head to tail
void displayList(ListDC head) {
    Node* temp = head;
    printf("List contents (Head to Tail): ");
    while (temp != NULL) {
        printf("%d ", temp->info);
        temp = temp->next;
    }
    printf("\n"); }

// Function to display the list from tail to head
void displayListReverse(ListDC tail) {
    Node* temp = tail;
    printf("List contents (Tail to Head): ");
    while (temp != NULL) {
        printf("%d ", temp->info);
        temp = temp->previous;
    }
    printf("\n");}

// Function to free the list memory
void freeList(ListDC head) {
    Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }}
int main() {
    ListDC head = NULL, tail = NULL;  // Head and tail of the list

    // Adding elements 4, 1, 5, 9 to the doubly linked list
    addNode(&head, &tail, 4);
    addNode(&head, &tail, 1);
    addNode(&head, &tail, 5);
    addNode(&head, &tail, 9);
```

```
// Display the list from head to tail
displayList(head);
// Display the list from tail to head (reverse order)
displayListReverse(tail);
// Free the list memory
freeList(head);
return 0;}
```

7. Special Linked Lists

7.1. Stacks

Definition

A **stack** is a linear data structure that follows the **Last In, First Out** (LIFO) principle. This means that the last element added to the stack is the first one to be removed. It works similarly to a stack of plates, where you can only add or remove the top plate.

Remarks

- Maintain a reference to the top of the stack.
- To push an element onto the stack, create a new node and point it to the current top, then update the top to the new node.
- To pop an element from the stack, remove the top node and update the top reference to the next node.

Main Operations

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the element from the top of the stack.
- **Peek** (or **Top**): Returns the element at the top of the stack without removing it.
- **IsEmpty**: Checks if the stack is empty.
- **Size**: Returns the number of elements in the stack.

7.2. Queues

Definition

A **queue** is a linear data structure that follows the **First In, First Out** (FIFO) principle. This means that the first element added to the queue is the first one to be removed. It works similarly to a line at a store, where people are served in the order **they arrived.**

Remarks

- Maintain references to both the front and rear of the queue.
- To enqueue (add) an element, create a new node and point the rear to it, then update the rear reference to the new node.
- To dequeue (remove) an element, remove the node at the front of the queue and update the front reference to the next node.

Main Operations

- **Enqueue**: Adds an element to the end of the queue.
- **Dequeue**: Removes the element from the front of the queue.
- **Front**: Returns the element at the front of the queue without removing it.
- **Rear**: Returns the element at the end of the queue.
- **IsEmpty**: Checks if the queue is empty.
- **Size**: Returns the number of elements in the queue.

Tutorial 1: Subprograms

Educational objectives: Decompose a complex problem into sub-problems. Differentiate between Procedures and Functions. Call a subroutine in a main program. Learn about passing parameters by value and by reference. Understand recursion.

Exercise 1

Write a parameterized action "Min" that finds the minimum of two integers. Write the main program that reads two integers, calls Min, and displays the result. (Provide two solutions, one using functions and the other using procedures).

Exercise 2

Consider a square matrix A (N, N) of integers (N≤25). Write a function to calculate the trace of matrix A. (The trace is the sum of the elements on the main diagonal.)

Exercise 3

Write a subroutine that determines if two words are anagrams. Two words are anagrams if they use the same letters. Examples: DOG is an anagram of GOD, CAT is an anagram of ACT.

Exercise 4

Write a subroutine "Facto" that calculates the factorial (n!) of an integer n.
Write a subroutine "Puiss" that returns the value x^n.
Write a subroutine "Expn" that approximates the value of e^x by calling the subroutines Facto and Puiss, such that:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Exercise 5

Consider a record E defined by two pieces of information: Position, an array of integers with a maximum of 100 elements, and N, an integer containing the number of elements in the Position array. Given a string M, write a procedure that returns a record of type E containing all positions of the substring 'ab' in the string M.
Example: M = "faabaababbaabrs", Positions: 3, 6, 8, 12, Number of elements N: 4

Exercise 6

Let "Tdate" be a date type composed of the integer fields DD, MM, YY.
- Write a PA "CompareD" to compare two dates D1 and D2.
- Let TD be an array of N dates (N≤100). Using the PA "CompareD", write an algorithm to sort this array in ascending order of dates.

Exercise 7

Write a function "Fibo" that calculates the Nth term (with N > 2) of the Fibonacci sequence. Define the Fibonacci sequence as follows:
U0=0, U1 = 1, Un = U(n-1) + U(n-2).
Write the recursive version of this function, "Fibo-rec".
Write the main program that calls one of the two functions.

Exercise 8

Write a recursive function to calculate the result of raising a number x to the power of y.

Exercise 9

Write a recursive subroutine that rearranges the elements of an array of character (string) in reverse order.

Tutorial 1 Solution

Exercise 1

Solution with Function

```
Algorithm: exercise-min;
Variables
x, y, p: integer

Function Min(a, b: integer): integer
VAR
res: integer
Begin
If a <= b Then
res ← a;
Else
res ← b;
End If
Min ← res
End

Begin
Write('Enter two integers x and y: ')
Read(x, y)
p ← Min(x, y)
Write('Minimum = ', p)
End
```

Solution with Procedure

```
Algorithm: exercise-min;
Variables
x, y, p: integer;

Procedure Min(a, b: integer, var min: integer)
Begin
If a <= b Then
min ← a;
Else
min ← b;
End If
End
```

```
Begin
Write('Enter two integers x and y: ')
Read(x, y)
Min(x, y, p)
Write('Minimum = ', p)
End
```

Exercise 2

```
Type Matrix = Array[1..25, 1..25] of integer

Function Trace(M: Matrix, N: integer): integer
Var Tr, I: integer
Begin
Tr ← 0
For I ← 1 to N Do
Tr ← Tr + M[I, I]
End For
Trace ← Tr
End
```

Exercise 3

```
Function IsAnagram(M1, M2: string): boolean
Var I, F1, F2: integer;
X: char;
IsAnag: boolean;
Begin
T1 ← Length(M1); T2 ← Length(M2);
If T1 ≠ T2 Then
IsAnag ← False
Else
IsAnag ← True; I ← 1;
While I ≤ T1 and IsAnag Do
X ← M1[I];
F1 ← 0;
For J ← 1 to T1 Do
If M1[J] = X Then
F1 ← F1 + 1
End If
End For
F2 ← 0;
```

For J ← 1 to T2 Do
If M2[J] = X Then
F2 ← F2 + 1
End If
End For
If F1 ≠ F2 Then
IsAnag ← False
End If
I ← I + 1;
End While
End If
IsAnagram ← IsAnag
End

Exercise 4

Function Fact(x: integer): integer
VAR
i, f: integer;
Begin
f ← 1;
For i ← 1 to x Do
f ← f * i;
End For
Fact ← f
End

Function Pow(x, n: integer): integer
VAR P, i: integer;
Begin
P ← 1;
If n = 0 Then
P ← 1;
Else
For i ← 1 to n Do
P ← P * x;
End For
End If
Pow ← P
End

Function Expn(x: real, n: integer): real
VAR exp, i: integer;
Begin
exp ← 1;

```
If n = 0 Then
exp ← 1;
Else
For i ← 1 to n Do
exp ← exp + (Pow(x, n) / Fact(n));
End For
End If
Expn ← exp
End
```

Exercice 5

```
Type RecordE = Record
Position: Array[1..100] of integer;
N: integer;
End

Procedure FindAbPositions(M: string, var Pos: RecordE);
Variable I, J, T: integer;
Begin
T ← Length(M); I ← 1; J ← 1; Pos.N ← 0;
While I < T Do
If M[I] = 'a' and M[I + 1] = 'b' Then
Pos.Position[J] ← I; Pos.N ← Pos.N + 1;
J ← J + 1;
I ← I + 2
Else
I ← I + 1
End If
End While
End
```

Exercise 6

```
Type Tdate = Record
 DD,MM,YY :integer
End

//We consider a Function that can take 1 For >, 0 For = and -1 For <

Function CompareD(D1,D2 :Tdate) :integer
begin
If D1.YY>D2.YY  Then
  CompareD←1
Else     If  D1.YY<D2.YY   Then
                    CompareD← -1
            Else  If  D1.MM>D2.MM  Then
```

```
                    CompareD ←1
                  Else If D1.MM<D2.MM Then
                         CompareD← -1
                         Else If D1.DD>D2.DD Then
                                     CompareD←1
                                Else If D1.DD<D2.DD Then
                                        CompareD← -1
                                        Else CompareD← 0
End if End if End if End if End if End if End if
End

Algorithm SortDate
Type Tdate = Record DD,MM,YY : integer  End
Var TD : Array[1..100] of Tdate
D : Tdate
I,J : integer
Function CompareD(D1,D2 : Tdate) : integer
begin
Write('Give N') ;
Repeat Read(N) ; Until N>0 and N≤100 ;
/*reading the date table
For I←1 to N Do
  Read(TD[I].DD, TD[I].MM, TD[I].YY)
End for
/*sorting
For I←1 to N-1 Do
For J←I+1 to N  Do
If CompareD( TD[I], TD[J])=1  Then
      D← TD[I]; TD[I]←TD[J]; TD[J]←D
End if;
End for
End for
/*display sorted dates
For I←1 to N Do
Write(TD[I].DD,'/', TD[I].MM,'/', TD[I].YY)
End for
End.
```

Exercise 7

```
Algorithm: Main
Variable x: integer;

Function Fibonacci(n: integer): integer
Var a, b, m: integer
```

```
Begin
a ← 1;
b ← 1;
m ← 1;
For i ← 2 to n Do
m ← a + b;
a ← b;
b ← m;
End For
Fibonacci ← m;
End

Function FibonacciRec(n: integer): integer
Begin
If n = 0 or n = 1 Then
FibonacciRec ← n;
Else
FibonacciRec ← FibonacciRec(n - 1) + FibonacciRec(n - 2);
End If
End

Begin
Write('Enter the term number, please: ');
Read(x)
Write('U= ', FibonacciRec(x) )
End
```

Exercise 8:

```
function power(x, y):
    if y == 0:
        return 1
    else:
        return x * power(x, y-1)
end
```

Exercise 9

```
Procedure Reverse-array (var arr: array of integer, var index: integer)
Var  int temp : integer
Begin
If (index <= length(arr) div 2) then
temp ← arr[index];
arr[index] ← arr[length(arr) – (index-1)]
arr[length (arr) – (index-1)] ← temp
 Reverse-array (arr, index+1)
```

End if
End
We should not perform a complete traversal of the array

Tutorial 2: Linked lists

Educational objectives: Understand Pointers. Learn about Dynamic Memory Management. Master Linked Lists use.

Exercise 1

Write a procedure that Create a list with N elements

Exercise 2

Write a subroutine that deletes a node with the given data key from the related Linked List

Exercise 3

Write the subroutine "insert-Node" that inserts a node containing the value X passed as a parameter

Exercise 4

Write the subroutine "delete-queue" that deletes the last node in the list.

Tutorial 2 Solution

Exercise 1

```
Algorithm CreateListhNElements
Variables
   Head, P : List
   N, i : Integer

Begin
   Write("Enter the number of elements:")
   Read(N)

   Head ← Nil  // Initialize the list as empty

   For i from 1 to N do
      Allocate(P)                  // Create a new node
      Write("Enter the value of element ", i, ":")
      Read(P^.Info)                // Read the value of the element

      P^.Next ← Head               // Insert at the head of the list
      Head ← P
   End For
End
```

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int info;
   struct Node* next;
} Node;

typedef Node* List;

void CreationListNElements(List* head) {
   List p;
   int N, i;

   printf("please enter the elements number: ");
   scanf("%d", &N);

   *Head= NULL; // empty list

   for (i = 1; i <= N; i++) {
      p = (Node*)malloc(sizeof(Node));
      if (p == NULL) {
         printf("Error\n");
         exit(1);
```

```
        }
        printf("Enter the element %d : ", i);
        scanf("%d", &(p->info));

        p->next = *head;
        *head = p;  }
}
```

Exercise 2

```
Procedure DeleteElement (Var Head: List, X: string)
Variables:
   P: List
   Prev: List     /* pointer to the element preceding the one to delete */
   Found: boolean
BEGIN
   IF Head ≠ Nil THEN   /* the list is not empty */
      IF Head^.info = X THEN  /* the element to delete is the first one */
         P ← Head
         Head ← Head^.Next
         Deallocate(P)
      ELSE  /* the element to delete is not the first one */
         Found ← False
         Prev ← Head      /* previous pointer */
         P ← Head^.Next   /* current pointer */
         WHILE P ≠ Nil AND NOT Found DO
            IF P^.Info = X THEN  /* the sought element is the current one */
               Found ← True
            ELSE  /* the current element is not the one we are looking for */
               Prev ← P         /* keep track of the previous node */
               P ← P^.Next      /* move to the next element in the list */
            END IF
         END WHILE
         IF Found THEN
            Prev^.Next ← P^.Next  /* skip over the element to delete */
            Deallocate(P)
         ELSE
            Write("The value ", X, " is not in the list")
         END IF
      END IF
   ELSE
      Write("The list is empty")
   END IF
END
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
```

```c
typedef struct Node {
    char info[100];          // Valeur de l'élément (chaîne de caractères)
    struct Node* suivant;    // Pointeur vers le prochain élément
} Node;

typedef Node* Liste;

void SupprimerElement(Liste* tete, const char* x) {
    Liste p, prec;
    bool trouve = false;

    if (*tete != NULL) { // La liste n'est pas vide
        if (strcmp((*tete)->info, x) == 0) { // L'élément à supprimer est le premier
            p = *tete;
            *tete = (*tete)->suivant;
            free(p);
        } else { // L'élément à supprimer n'est pas le premier
            prec = *tete;
            p = (*tete)->suivant;
            while (p != NULL && !trouve) {
                if (strcmp(p->info, x) == 0) { // Élément trouvé
                    trouve = true;
                } else {
                    prec = p;
                    p = p->suivant;
                }
            }
            if (trouve) {
                prec->suivant = p->suivant; // On "saute" l'élément à supprimer
                free(p);
            } else {
                printf("La valeur \"%s\" n'est pas dans la liste\n", x);
            }
        }
    } else {
        printf("La liste est vide\n");
    }
}
```

Exercise 3

```
Procedure InsertAtHead (Head: var List, X: String)
   Variable P: Node
Begin
      Allocate (P)
      P^.Info ← X
      P^.Next ← Head
      Head ← P
End Procedure
```

Exercise 4

```
Procedure DeleteLastNode (Head: Var List)
   Variable P, Prev: Node
begin
   // Case 1: List is empty
   If Head = Nil Then
      Print "List is empty."

   End If

   // Case 2: List has only one node
   If Head^.Next = Nil Then
      Deallocate(Head)
      Head ← Nil
   End If

   // Case 3: List has more than one node
   P ← Head
   While P^.Next ≠ Nil Do
      Prev ← P
      P ← P^.Next
   End While
   Prev^.Next ← Nil
   Deallocate(P)
End Procedure
```

Bibliographic References

[1]University of Nantes, "Algorithmics."
http://miage.univ-nantes.fr/miage/DVD-MIAGEv2/Algo.html
(accessed on June 12, 2023).

[2] P. Geurts, "Data Structures and Algorithms."

[3] S. Akrouf, " Informatique générale " University of Bordj Bou Arréridj, 2012.

[4] M. F. Omar, "Algorithm and Programming Course," University of Sciences and Technology in Oran, Working Paper, May 2019. Accessed on: June 12, 2023. [Online]. Available at: http://dspace.univ-usto.dz/handle/123456789/380

[5] "Les algorithmes pour les Nuls grand format - Google Books." https://www.google.dz/books/edition/Les_algorithmes_pour_les_Nuls_grand_form/7uUzDwAAQBAJ?hl=fr&sa=X&ved=2ahUKEwj5z9HKt73_AhVNVKQEHQx7DQQQiqUDegQIERAH (accessed on June 12, 2023).