

STRINGS

1. Definition

A string object consists of a sequence of characters placed between apostrophes. This sequence can be empty ("), of length 1 (i.e., a single character), or of length greater than 1 (up to a maximum of 255 characters).

Example

'TOTO' is a string of length 4.

2. Declaration

- A string can be of predefined size (up to 255 characters).

Syntax

VAR

Identifier1, Identifier2, ..., IdentifierN: string

- A limit size can be specified for a string.

Syntax

VAR

Identifier1, Identifier2, ..., IdentifierN: string [size]

With $0 < \text{Size} \leq 255$

3. Manipulation Operations

The permitted operations on strings are as follows:

- Reading / Writing / Assignment: These actions can be performed globally for the entire string.

Example

Read (ch): to read the entire string ch.

- **LEN:** This function returns the length of the string expressed in number of characters. The result of this function is of integer type.

Syntax

LEN (string)

- Comparisons ($=$, $<$, $>$, \leq , \geq): they are based on lexicographical order by comparing the characters in the same position of the two strings. The two strings are compared from left to right, character by character. The comparison sign is deduced from the first difference.

Examples

'BALLES' < 'BALLON' < 'BAR' < 'Bar' < 'bar'

'BAL' < 'BALLES'

- **CONCAT:** it allows forming a single string from two strings placed next to each other in the order of their appearance. The result of this function is a string of maximum size 255 characters.

Syntax

CONCAT (ch1, ch2)

Example

CONCAT ('AUTO', 'BUS') returns 'AUTOBUS'

- **SCH:** It allows extracting a substring from another string.

Syntax

SCH (ch, P, L): allows extracting, from a string ch, a substring of length L starting from position P.

Example

SC1 \leftarrow SCH ('AUTOBUS', 1, 4): SC1 will have the value 'AUTO'

SC2 \leftarrow SCH ('AUTOBUS', 5, 3): SC2 will have the value 'BUS'

Note

If $P + L - 1 > \text{LEN}(\text{ch})$, then we will have a string of length $\text{LEN}(\text{ch}) - P + 1$.

Example

SC1 SCH ('AUTOBUS', 5, 4): SC1 will have the value 'BUS'

- POS: allows finding the position of the first occurrence of one string within another.

Syntax

POS (ch, ch1): Returns the position of the first occurrence of string ch1 in string ch. It returns a value n (with $0 < n \leq \text{LEN}(ch)$) if ch1 exists in ch, 0 otherwise.

Application 1

Write an algorithm that allows reading a non-empty string and displaying its length.

Application 2

Write an algorithm that allows:

- reading a non-empty string,
- displaying the character at a given position i,
- splitting the string into two strings from position i.

Application 3

Write an algorithm and the necessary sub-algorithms that allow:

- reading a string of characters.
- reading a character C and calculating the number of occurrences of the character C in the string.

Application 4

Write an algorithm and the necessary sub-algorithms that allow:

- reading a string of alphabetical characters.
- converting the string to lowercase.

STRUCTURES

1. Definition

A structure is a grouping of data consisting of a fixed number of elements called fields, each having a name and a type.

2. Declaration

Syntax

TYPE

```
record_name = Structure
<Field_Name1>: type1
<Field_Name2>: type2
...
<Field_NameN>: typeN
End Structure
```

VAR

```
variable_name: record_name
```

With:

Field_Namei: represents the name of the "i" field of the structure.

typei: represents the type of the "i" field. It can be simple or composite.

Example

Suppose we want to manage all the information of a student known by their Last name, first name, average, and age. The corresponding structure will be as follows:

TYPE

```
Student = Structure
LastName, FirstName: string[30]
Average: real
Age: integer
```

End Structure

VAR

Stud: Student

3. Manipulation Operations

- Accessing an element of a Structure type must follow the following syntax:

variable_name•field_name

Example

Stud•Average: corresponds to the average of the student Stud.

Stud•LastName: corresponds to the last name of the student Stud.

- Reading/Writing/Assignment: A structure must be handled field by field and not as a single entity.

- Each field can undergo the allowed functions for its type.

Application 1

Write an algorithm to input the information of a student (Considering the Student structure defined previously).

Application 2

Give the data structures to represent a student; knowing that a student is characterized by their last name, first name, date of birth (day, month, and year), and their overall average.

TYPE

Date = Structure

DD: integer

MM: integer

YY: integer

End Structure

Student = Structure

lastName, firstName: string[30]

dateOfBirth: Date

average: real

End Structure

VAR

Stud: Student

Application 3

If we want to record the averages of 10 subjects for each student, as well as their overall average, the Student structure will be as follows:

TYPE

Student = Structure

LastName, FirstName: string[30]

DateOfBirth: Date

AverageSubjects: array [1 .. 10] of real

OverallAverage: real

End Structure

RECURSION

I. Concept of recursion

Recursion is a concept that is frequently encountered in everyday life: arrays within arrays, nested dolls, etc.

In computer science, recursion can involve processing: A recursive process is one that calls itself (function/procedure).

Examples:

- **Definition of a mathematical function $f(x)=f(x-1)+4$**
 $f(0) = 4$
- **Definition of a character string: character + a string**
a character: letter|digit|empty string
- **Recursive definition of the factorial function: $N! = N \times (N-1) \times \dots \times 1$**
 - (i) $0! = 1$.
 - (ii) For N a natural number ($N > 0$), $N! = N \times (N - 1)!$.

II. Recursion concept

In any recursive definition, there must be:

- a certain number of cases whose resolution is known, these "simple cases" will form the termination cases of recursion, called the base of recursion;
- a way to reduce from a "complicated" case to a "simpler" case.

Difficulties: It must be ensured that one always falls back on a known case, that is, on a termination condition; it must then be ensured that the function is completely defined, that is, it is defined over its entire domain of application.

Example: The factorial function

The factorial function ($n! = n \times (n-1) \times (n-2) \times \dots \times 1$) is usually defined as follows:

- (i) factorial (n) = 1 for $n = 0$;
- (ii) factorial (n) = $n \times$ factorial ($n - 1$) for $n > 0$.

We obtain the following recursive function for calculating $N!$:

```

Function Fact (N: integer): integer
VAR
  F: integer
BEGIN
  IF (N=0) Then
    F ← 1
  ELSE
    F ← n* Fact (n-1)
  END IF
  Fact ← F
END Fact
    
```

III. Types of Recursive Algorithms

An algorithm is said to be recursive when it calls itself in its definition (direct recursion), and potentially when it calls itself through other algorithms called by itself (indirect recursion).

3.1.Direct Recursion

In the case of direct recursion, the recursive call of a sub-algorithm P occurs within the sub-algorithm P itself. This call can be made once (simple recursion) or multiple times (multiple recursion).

- **Simple Recursion**

```
Sub-Algorithm P( )
...
BEGIN
...
P // direct recursive call of P within the body of P
...
END
```

Example:

The power function $f(x) = x^n$ can be recursively defined as follows:

$x^n = 1$ if $n = 0$;

$x^n = x \times x^{n-1}$ if $n \geq 1$.

The corresponding algorithm is written as follows:

```
Function POWER(x, n: integer): integer
VAR
  P: integer
BEGIN
  IF n = 0 then
    P ← 1
  ELSE
    P ← x × POWER(x, n-1)
  END IF
  POWER ← P
END POWER
```

- **Multiple Recursion**

A recursive definition can contain more than one recursive call.

```
Sub-Algorithm P( )
...
BEGIN
...
P // direct recursive call of P within the body of P
...
P // direct recursive call of P within the body of P
....
END
```

Example:

Here we want to compute the combinations C_n^p using the following relationship:

$$C_n^p = \begin{cases} 1 & \text{if } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{else.} \end{cases}$$

```

Function COMBINATION(n, p: integer): integer
VAR
  C: integer
BEGIN
  IF (p = 0 or p = n) then
    C ← 1
  ELSE
    C ← COMBINATION(n-1, p) + COMBINATION(n-1, p-1)
  END IF
  COMBINATION ← C
END COMBINATION

```

3.2. Indirect Recursion

Sometimes, it is possible for a sub-algorithm P to call a sub-algorithm Q, which in turn calls the sub-algorithm P. This is called indirect recursion.

Sub-Algorithm P()	Sub-Algorithm Q()
<pre> ... BEGIN ... Q //indirect recursive call of P within the body of Q ... END P </pre>	<pre> ... BEGIN ... P //direct recursive call of Q within the body of P ... END Q </pre>

The procedure P uses the procedure Q for its definition, which itself uses P to define itself.

Example: A simple example of indirect recursion is the recursive definition of even and odd numbers. A positive number n is even if n-1 is odd; a positive number n is odd if n-1 is even.

$$\text{even}(n) = \begin{cases} \text{True} & \text{if } n = 0; \\ \text{odd}(n-1) & \text{else;} \end{cases}$$

$$\text{odd}(n) = \begin{cases} \text{False} & \text{if } n = 0; \\ \text{even}(n-1) & \text{else;} \end{cases}$$

The corresponding algorithms are written as follows:

```

Function Even (n : integer) : Boolean
VAR
  L : Boolean
BEGIN
If n = 0 then L ← True
Else L ← Odd (n-1)
END If
Even ← L
END Even

```

```

Function Odd (n : integer) : Boolean
VAR
  L : Boolean
BEGIN
If n = 0 then L ← False
Else L ← Even (n-1)
END If
Odd ← L
END Odd

```

IV. Functioning of a Recursive Algorithm

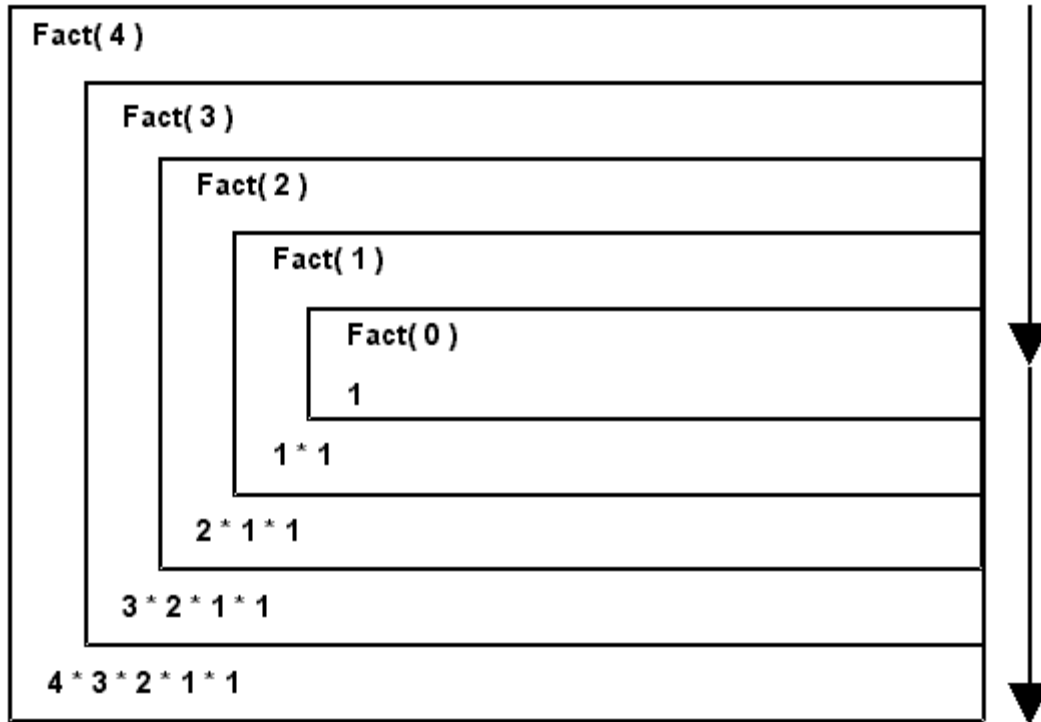
Example 1: The factorial function

Let's consider the factorial function Fact as defined in paragraph 2.

Upon calling:

$Y \leftarrow \text{Fact}(4)$

The execution of the Fact function can be described by the following figure:



DYNAMIC STRUCTURES

If we work with a variable of type array of size N , we can find ourselves in one of the following two situations:

- The reserved size is very large compared to the number of occupied elements \rightarrow we will have extra, unused lost cells.
- The reserved size is smaller than the number of elements we want to manage \rightarrow we will have a shortage of elements.

It is then a matter of finding an idea that allows defining structures of undetermined sizes \rightarrow hence the type of dynamic objects.

1. Notions of Dynamic Objects

Dynamic variables are created, modified, and deleted as needed during the execution of an algorithm.

Dynamic variables are not declared in the variable declaration section, and they cannot be referenced by a direct identifier. They are referenced through a special variable called a Pointer. A pointer is a static variable that contains the memory address of another variable (dynamic). Its role is to locate, reference, or designate another object.

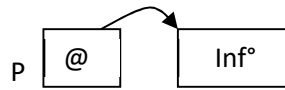
A dynamic variable is an object created during the execution of a program. It has a type and a value but no name.

A pointer type is defined by the symbol \uparrow followed by the type of the dynamic variable.

Example:

P: empty pointer

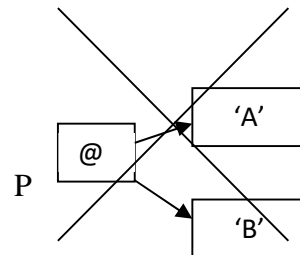
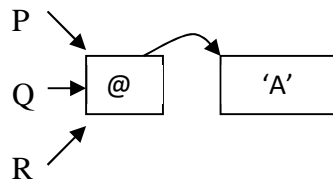
P: pointer pointing to a cell containing information



with P: being a static object of type address

$P\uparrow$: integer, real, char, etc. \rightarrow is an element designated by the pointer P.

Note: an object of dynamic type can be pointed to by one or more pointers. Whereas a single pointer can only point to a single object.



2. Manipulation of Dynamic Objects

2.1. Declaration of a Pointer

Syntax:

P: \uparrow base-type


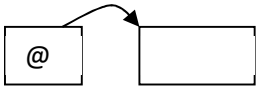
Example:

- Declaration of a pointer to an integer
P: \uparrow integer
- Declaration of a pointer to a character
Q: \uparrow char

2.2. Creation of a Dynamic Object

The creation action is expressed by the action Allocate(P).

If space is available, then P will take the address of the created (dynamic) variable.

Initial state	Action	Final state
P 	Allocate (P)	P 

2.3. Access to a Dynamic Object

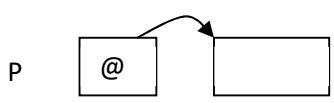
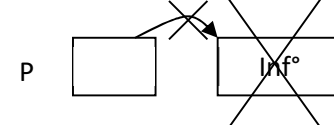
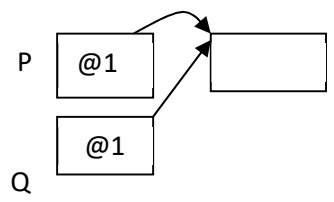
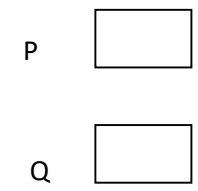
Access to a dynamic object referenced by a pointer P is done by $P\uparrow$.

Application: Write a sub-algorithm to:

- create a dynamic variable of string type and enter a value into it,
- display the length of the string.

2.4. Deletion of a Dynamic Object

The deletion action is expressed by the action Free(P).

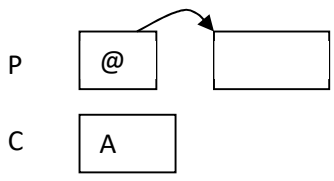
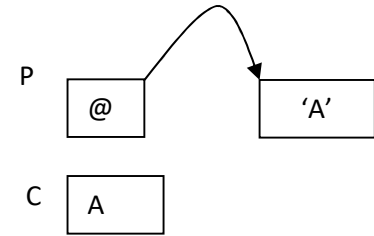
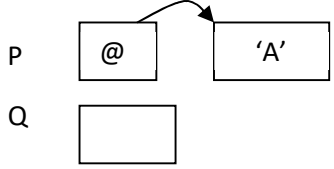
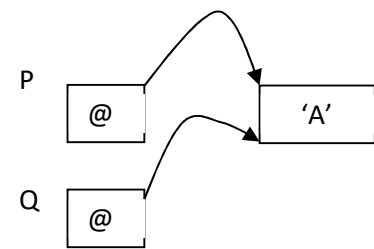
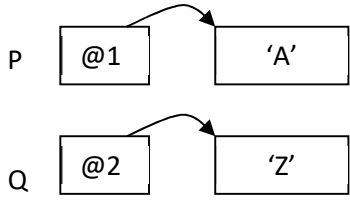
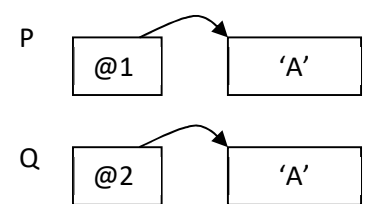
Initial state	Action	Final state
	Free (P)	
	Free (P)	

Freeing a pointer involves deleting the dynamic variable referenced by the pointer P and not the pointer itself.

2.5. Other operations

Let P be a pointer pointing to an element of dynamic type. Let Q be another pointer.

If we want Q to point to the same variable as pointed to by P; then we must perform the action:
 $Q \leftarrow P$.

Initial state	Action	Final state
	$P \uparrow \leftarrow 'A'$ or $P \uparrow \leftarrow C$	
	$Q \leftarrow P$	
	$Q \uparrow \leftarrow P \uparrow$	

3. Singly Linked Lists

3.1. Definition

It is a set of dynamic objects linked together by pointers. The list is defined by the pointer to the first object.

Singly linked lists are used to construct a data structure whose size is not known in advance.

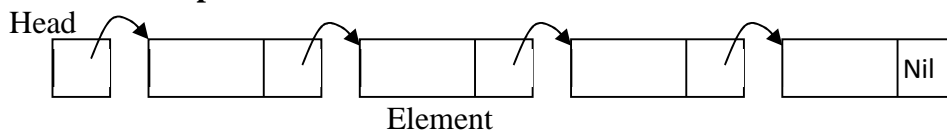
3.2. Characteristics

A list is characterized by:

- A head element that denotes the first element of the list.
- Every element of a list consists of two parts:
 - The first part contains one or more fields that form the information of the elements to be processed.
 - The second part contains a field allowing the link to the next element in the list. This field is a 'pointer'.

Note: The link field of the last element of the list contains Nil (Nothing In List) which signifies the end of the list.

3.3. Schematic Representation



3.4. Declaration

Syntax:

```

TYPE
    ListItem = Structure
        Information(s): base_type(s)
        Next: ListItem
    End Structure
VAR
    Head: ListItem
    
```

where:

<Information(s)> represents the set of information of an element.

<base_type(s)> represents the basic types of all information, which can be simple or composite, predefined or custom.

Next is a pointer field that points to the next object.

Example: If we want to manage the information of a group of people whose names and ages are known. The necessary structure will then be as follows:

```

TYPE
    Person = Structure
        Name: string[10]
        Age: integer
        Next: Person
    End Structure
    
```

```

VAR
    P: Person
    
```

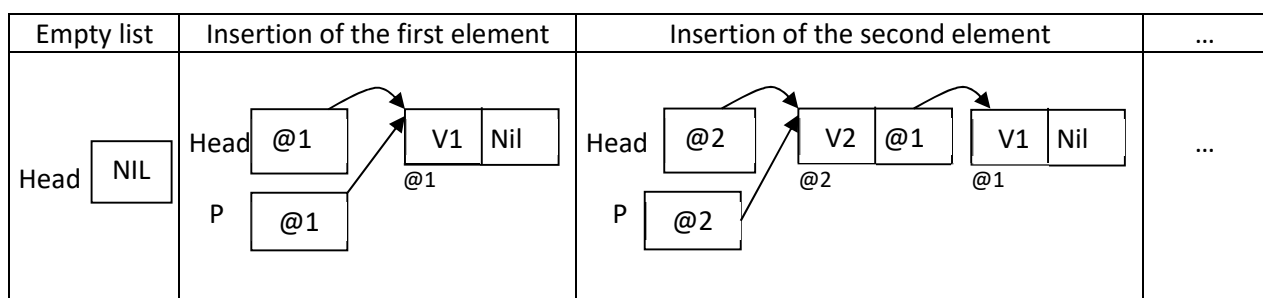
3.5. Manipulation Operations

➤ Creating a singly linked list by adding at the head

The following algorithm creates a singly linked list of integers by adding elements at the head as many times as the user desires.

```

ALGORITHM create_head
TYPE
list_item = Structure
    Info: integer
    Next: list_item
End Structure
VAR
    Head, P: list_item
    Resp: char
BEGIN
    Head  $\leftarrow$  Nil
    Repeat
        Allocate (P)
        Read (P .Info)
        P .Next  $\leftarrow$  Head
        Head  $\leftarrow$  P
        Write ("Add another? Y/N")
        Read (Resp)
    Until (Resp = 'N')
END
    
```



➤ **Creating a singly linked list by adding at the tail**

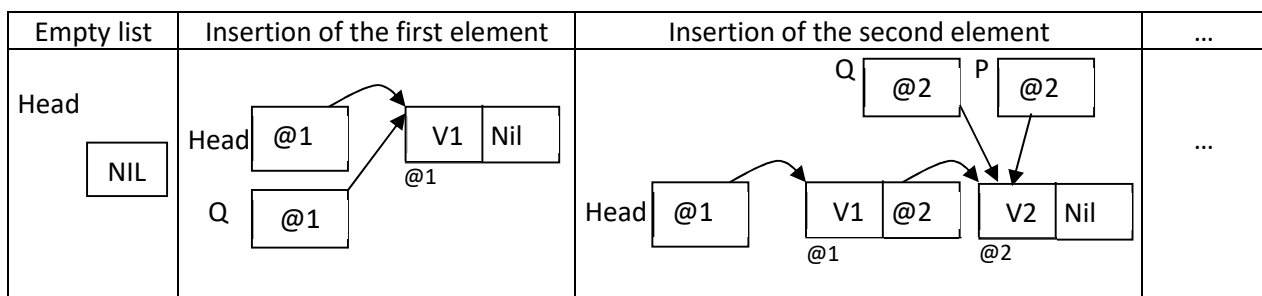
```

ALGORITHM create_tail
TYPE
list_item = Structure
    Info: integer
    Next: list_item
End Structure
VAR
    Head, P, Q: list_item
    Resp: char
BEGIN
    Head  $\leftarrow$  Nil
    Q  $\leftarrow$  Head
    Repeat
        Allocate (P)
        Read (P .Info)
        P .Next  $\leftarrow$  Nil
        If (Head = Nil) Then
            Head  $\leftarrow$  P
    
```

```

Else
    Q .Next ← P
End If
Q ← P // Q contains the address of the last element in the list
Write ("Add another? Y/N")
Read (Resp)
Until (Resp = 'N')
END

```



➤ Traversing a List

If we have a singly linked list and we want to display all the information it contains, we must traverse the list element by element.

To move a pointer P (which points to a given element) to the next element, we must use the action $P \leftarrow P .\text{Next}$.

Application:

Write a procedure to display the elements of a singly linked list pointed to by a pointer Head.

Procedure Display_List(**INPUT** Head: list_item)

Var

Ptr : list_item

Begin

Ptr ← Head

While (Ptr < > Nil) **do**

Write(Ptr .Info)

Ptr ← Ptr .Next

End While

End

➤ Adding to a List

1. Adding to the Head of a List

If we want to add an element to the head of an already created list, its Next field must point to the first element of the list. The head must then change to point to the new element to be inserted.

Application: Write a procedure to add an element to the head of a list.

Procedure Add_To_Head(**IN** NewItem: integer ; **INOUT** Head: list_item)

Var

P : list_item

Begin

Allocate (P)

P .Info ← NewItem

P .Next ← Head

Head ← P

End

2. Adding to the Tail of a List

If we want to add an element to the tail of the already created list, we must proceed as follows:

- Create the element to add and set its Next field to Nil.
- Traverse the list and position ourselves on the last element.
- Perform the necessary chaining.

Application: Write a procedure to add an element to the tail of the list.

Procedure Add_To_Tail(**IN** NewItem: integer ; **INOUT** Head: list_item)

Var

Ptr,P : list_item

Begin

Allocate (P)

P .Info \leftarrow NewItem

P .Next \leftarrow Nil

If (Head = Nil) Then

Head \leftarrow P

Else

Ptr \leftarrow Head

While (Ptr .Next \neq Nil) do

Ptr \leftarrow Ptr .Next

End While

Ptr .Next \leftarrow P

End If

End

➤ Deleting an element from a list

1. Deleting the head element

To delete the Head element of the list, we must use an intermediary pointer to store the address of the head, move the head pointer to the next element, and free the intermediary pointer.

Application 1: Write a procedure to delete the first element of a list

Procedure Delete_From_Head(**INOUT** Head: list_item)

Var

P : list_item

Begin

If (Head \neq Nil) Then

P \leftarrow Head

Head \leftarrow Head .Next

Free(P)

End If

End

2. Deleting the last element

To delete the last element of the list, we must traverse the list until we locate the last element and its predecessor.

Application 2: Write a procedure to delete the last element of a list

Procedure Delete_From_Tail(**INOUT** Head: list_item)

Var

BP, P: list_item

Begin

If (Head \neq Nil) Then

If (Head .Next = Nil) Then

```

    P ← Head
    Head ← Nil
Else
    BP ← Head
    P ← Head .Next
    While (P .Next < > Nil) do
        BP ← P
        P ← P .Next
    End While
    BP .Next ← Nil
End If
Free(P)
End If
End

```

4. Doubly Linked Lists

4.1. Definition

Sometimes, there is a need to traverse a list backwards or to provide the previous and next elements of a given element in a list. To achieve this, we can add a second pointer in each element of the list that will point to the element preceding a given element.

4.2. Declaration

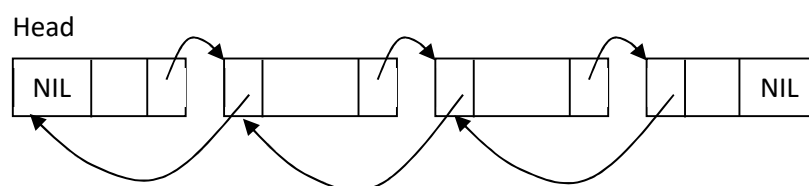
Syntax:

```

TYPE
    ListItem = Structure
        Previous: ListItem
        Information(s): base_type(s)
        Next: ListItem
    End Structure
VAR
    Head: ListItem

```

4.3. Schematic Representation



Application 1: Write a procedure to add an element at the head of a doubly linked list.

Procedure Add_To_Head(**IN** NewItem: integer ; **INOUT** Head: list_item)

Var

P : list_item

Begin

Allocate (P)

P .Info ← NewItem

P .previous ← NIL

If (Head = NIL) then

```

    P .Next ← NIL
Else
    P .Next ← Head
    Head .previous ← P
End if
Head ← P
End

```

Application 2: Write a procedure to add an element at the tail of a doubly linked list.

Application 3: Write a procedure to add an element to a doubly linked list after an element pointed to by a pointer Q.

Application 4: Write a procedure to delete any element from a doubly linked list assumed to be created. The element to be deleted is pointed to by a pointer P.



5. Stacks

5.1. Definition

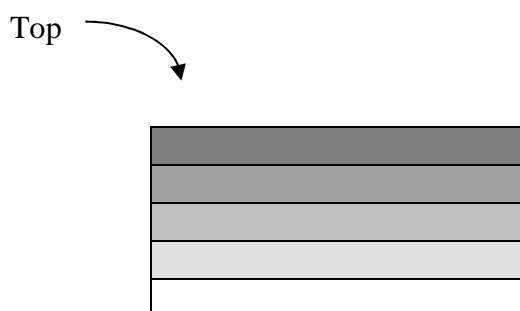
A stack is a collection formed of a variable, possibly zero, number of dynamic data elements. A stack is based on the principle of "last in, first out" (or LIFO for Last In, First Out), which means that the most recently added elements to the stack will be the first to be retrieved. The operation is similar to that of a stack of plates: plates are added to the stack, and they are retrieved in reverse order, starting with the last one added.

5.2. Examples of stack usage

The concept of a stack is involved in many computer problems. For example:

- In a web browser, a stack is used to memorize the visited web pages. The address of each new visited page is stacked, and the user pops the address of the previous page by clicking the "Back" button. 
- The "Undo" function  in a word processor memorizes the changes made to the text in a stack.
- Particularly for recursive problems, an implicit call stack is used.

5.3. Schematic representation of stacks



5.4.Stack Declaration

TYPE

Stack_Element = Structure

Info : Element_Type

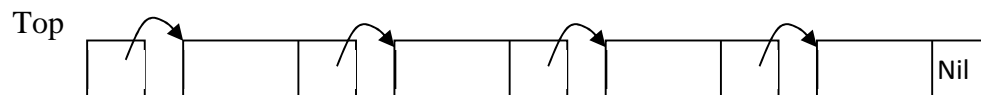
Next : Stack_Element

End Structure

VAR

TOP : Stack_Element

5.5.Memory Representation of Stacks



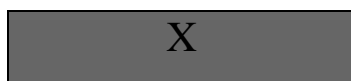
5.6.Stack Manipulation Operations

A stack of objects is characterized by the following operations:

- **CREATE_STACK**: creation of an empty stack,
- **STACK_EMPTY**: test if the stack is empty,
- **PUSH**: adding an element to the stack,
- **POP**: removing an element from the stack,

Example: Adding an element to a stack.

Initial State

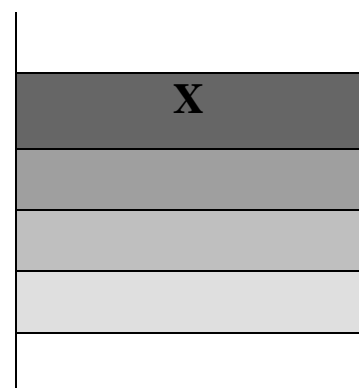


Action

Push (X)



Final State



Application: Write the procedures and functions to perform:

- Creating a stack,
- Adding an element to the stack,
- Removing an element from the stack,
- Testing if the stack is empty or not,
- Calculating the number of elements in the stack.

procedure create(Output top: Stack)

Begin

top \leftarrow nil

end create

procedure push(Input val: integer; InOut top: Stack)

VAR

new: Stack

Begin

Allocate(new)

new .info \leftarrow val

new .next \leftarrow top

top \leftarrow new

end push

procedure pop(Output info: integer; InOut top: Stack)

VAR

p: Stack

Begin

info \leftarrow top .info

p \leftarrow top

top \leftarrow top .next

free(p)

end pop

function is_empty(top: Stack): boolean

Begin

is_empty \leftarrow top = nil

end is_empty

function count_recursive(top: Stack): integer

Begin

if (top = nil) then

count \leftarrow 0

else

count \leftarrow 1 + count(top .next)

end if

end count_recursive

function count_iterative(top: Stack): integer

VAR

length: integer

current: Stack

Begin

length \leftarrow 0

current \leftarrow top

while (current \neq nil) do

length \leftarrow length + 1

current \leftarrow current .next

```

    end do
    count  $\leftarrow$  length
end count_iterative

```

6. Queues

6.1. Definition

A queue is a collection formed of a variable, possibly zero, number of dynamic data elements. A queue is based on the FIFO (First In, First Out) principle, which means that the first elements added to the queue will be the first to be retrieved. The operation is similar to a waiting line: the first people to arrive at a counter are the first to leave the queue.

6.2. Example of queue usage

Queues are applied in various computer science domains, including:

- The domain of operating systems and process control, where events can occur at any time and must be processed in their arrival order.
- Print servers, which must process requests in the order they arrive and insert them into a queue.

6.3. Queue Declaration

```

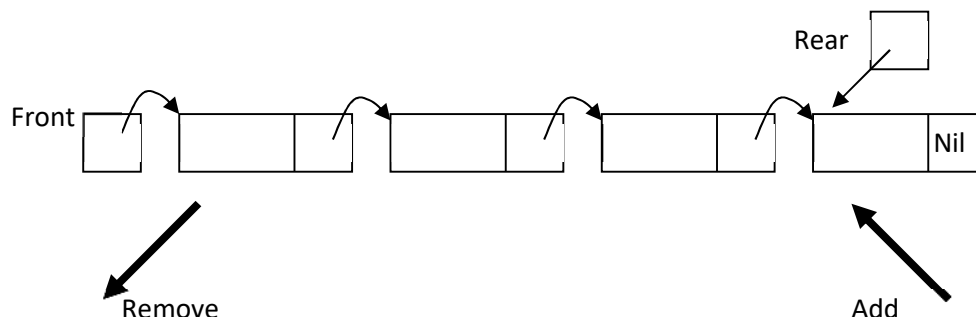
TYPE
    Queue_Element = Structure
        Info : Element_Type
        Next : Queue_Element
    End Structure
    Queue = Structure
        Front, Rear : Queue_Element
    End Structure
VAR
    Q : Queue

```

6.4. Schematic representation of a queue



6.5. Memory representation of a queue



6.6. Queue Manipulation Operations

A queue of objects is characterized by the following operations:

- CREATE_QUEUE: creation of an empty queue,

- QUEUE_EMPTY: test if the queue is empty,
- ENQUEUE: adding an element to the queue,
- DEQUEUE: removing an element from the queue,

Application: Write the procedures and functions to perform:

- Creating a queue,
- Adding an element to the queue,
- Removing an element from the queue,
- Testing if the queue is empty or not,
- Calculating the number of elements in the queue.

procedure create(Output q: Queue)

Begin

q .front \leftarrow nil

q .rear \leftarrow nil

end create

procedure enqueue(Input val: integer; InOut q: Queue)

VAR

new: QueueEl

Begin

Allocate(new)

new .info \leftarrow val

new .next \leftarrow nil

if (q .front = nil) then

q .front \leftarrow new

else

(q .rear) .next \leftarrow new

end if

q .rear \leftarrow new

end enqueue

procedure dequeue(Output info: integer; InOut q: Queue)

VAR

temp: QueueEl

Begin

temp \leftarrow q .front

info \leftarrow temp .info

if (q .front = q .rear) then

q .front \leftarrow nil

q .rear \leftarrow nil

else

q .front \leftarrow temp .next

end if

free(temp)

end dequeue

function is_empty(q: Queue): boolean

Begin

is_empty \leftarrow q .front = nil AND q .rear = nil

end is_empty

Strings and Structures

Exercise 1

Write a function PALINDROME that checks if a given string is a palindrome.

Example: 'RADAR' is a palindrome.

Exercise 2

1. Write a function OCCURRENCE that calculates the number of occurrences of a character C in a string CH.
2. Write a procedure FILL_A that fills an array of size N ($N \leq 100$) with strings.
3. Write a procedure DISPLAY that displays an array of strings.
4. Write a function OCC_ARRAY that calculates the number of occurrences of a character C in an array of strings using the OCCURRENCE function.
5. Write an algorithm that:
 - Fills two arrays T1 and T2 of the same size with strings.
 - Inputs a character C.
 - Displays the array that has the most occurrences of the character C. (In case of a tie in the number of occurrences, display both arrays).

Exercise 3

Write a function APPEARANCE that calculates the number of occurrences of a given first string in a second string.

Exercise 4

Let "OBJETINFO" be a structure type formed of the fields (Name, Size, and Type) as indicated in the example above:

Nom	Taille	Type
employee.pas	1	Fichier PAS
Informatique.doc	16	Document Microsoft...
TPCW.exe	72	Application
TPW.pas	498	Application
BORTE.gif	12	Fichier de police
employee.exe	9	Application

Write an algorithm that calls the necessary sub-algorithms and:

- loads an array T of N elements of type OBJETINFO,
- finds the largest object in T,
- finds the total size of the objects in T,
- displays objects with the extension ".exe".

Exercise 5

1. Write a function that checks the validity of a given date in the form of a structure composed of day, month, and year.
2. Write a sub-algorithm that allows entering a valid date (in the form of a day/month/year structure).
3. Write a sub-algorithm that displays a date.
4. Write a sub-algorithm that compares two dates and returns the most recent one.
5. Write an algorithm that allows reading two dates D1 and D2 and displays the most recent one between them.

Recursion

Exercise 1

Write a recursive function NBDigits that calculates the number of digits in a positive integer.

Example: for $N = 4586$: the result is 4

Exercise 2

Write a recursive function SomDigits that calculates the sum of the digits of a positive integer.

Example: for $N = 4586$: the result is 23

Exercise 3

Write a recursive procedure to reverse a given string of characters.

Exercise 4

Write a recursive function to check if a given string of characters is a palindrome or not.

Exercise 5

Write a recursive procedure that displays the elements of an integer array T of size N .