Chapter 4

Sequential Structures

4.1 Introduction to dynamic memory allocation

4.1.1 Problem of static arrays

Indeed, working with static arrays in algorithms allows us resolving several kinds of problems. This was presented in the three first chapters of the course. However, static arrays suffer from two basic disadvantages:

- They are rigid: this means that the size of the array must be predefined before the compiling of the program. Thus, we have to define the static size of the array, then this size can no more change during the execution of the program. To update the size, we have to stop the execution of the program then re-edit the program and so on. It is clear that this situation cause un-efficiency of the programming, because it is possible that we don't know previously the suitable required size, so that we can be in a situation where an extra size is predefined then it will be not used (losing unused memory during the execution of the program), or the predefined size is less than the required memory which makes the execution of the program impossible. As an example, suppose we want to solve the following problem: "Find all prime numbers from 1 to n and store them in memory". The problem lies in the choice of the reception structure. If we use an array, it is not possible to define the size of this array precisely even if we know the value of n (for example 10000). We are therefore, here, faced with a problem where the reservation of space must be dynamic.

- They are contiguous: Besides being rigid with an unchangeable size, static arrays are supposed to be contiguous. That is to say that each element of index i immediately follows in memory the previous element of index $i-1$. This means that the sequence of array cells must be recorded in a sequence of free cells in the machine's memory. In this case, it will be not possible to execute the program unless we have enough contiguous free space memory to save our arrays.

4.1.2 How to resolve the problem

It is clear that we are looking for a dynamic allocation of the memory instead of static, predefined and unchangeable reservation. We are interested to provide the programmer with the ability to reserve (or allocate) memory dynamically, according to his needs. The programmer should have also the ability to release (or free) the unused memory at any time. This dynamic allocation and release of the memory will be realized during the execution of the program, ensuring more efficient management of the memory and allowing the execution of any kinds of programs even the required memory space is unknown previously. To ensure this property, we introduce a new kind of type (**Pointers**) with new set of *operations* in the next item of this section.

4.1.3 Introducing the Pointer type

1. Definition: a Pointer is a variable which has a specific value. This value is an address (or a reference) that points (or refers) to another memory area (a memory cell, for example) which contains another value belonging to some other type (integer, float, string, or even another Pointer). Figure 4.1.3 shows this situation.

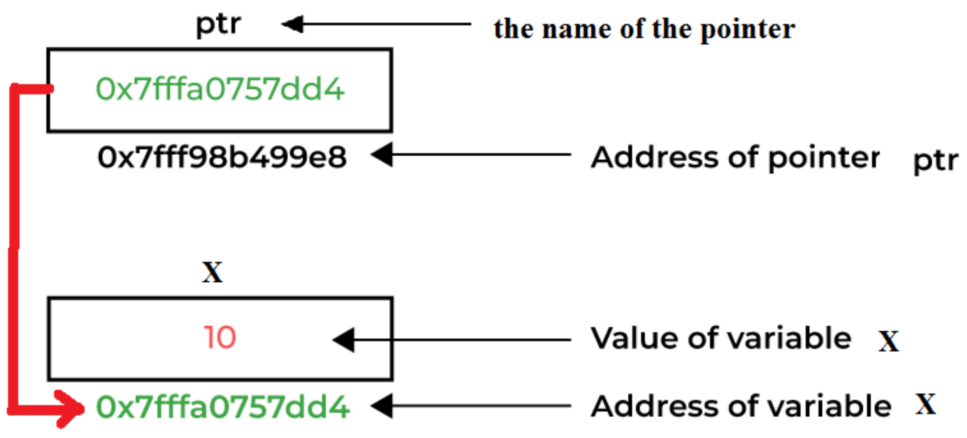


Figure 4.1: A pointer and the pointed value

In this figure, we have a pointer called **ptr** that is created and saved in a memory cell with the address 0x7fff98b499e8. This pointer **ptr** refers (or points) to another memory cell (address: 0x7fffa0757dd4) which contains an integer value called X and with the value 10. As showed, the address of **X** represents the value of **ptr** and we say that pointer **ptr** points variable **X**. Next, we will present the set of operations to be used on the type pointer.

2. Operations:

Three kinds of operations are possible:

- The **declaration** of the pointer: This is done as any other variable declaration. It is done as follows:
  - **var** p:***Pointer***; //In this case, p is a general pointer that can point on all types of variables.
  - **var** p:***Pointer***(*Basic_ Type*); Basic_Type: is the type of variables that will be pointed by p. For example : **var** p:***Pointer***(integer); this means a pointer variable **p** that points to an integer variable. We can use also the syntax of C language as follows: ***int* p;*** In this declaration, we have the type ***int*** for integers and the type ***int**** for pointers on integer.

  On Figure 4.1.3, we show the effect of the declaration of a pointer. As we see, there is only a memory for the pointer variable, but without any content currently. The pointer is not yet created, it does not point to anything.

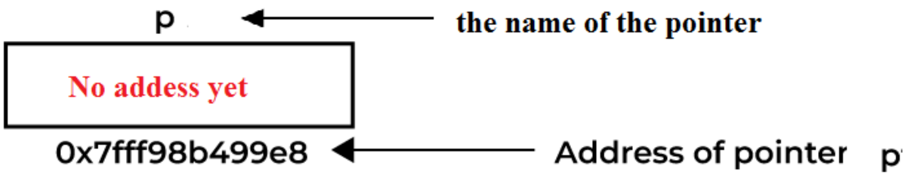


Figure 4.2: A declared pointer.

- The **allocation** of the pointer: We have to understand that the previous declaration of a pointer variable p reserves only the memory where p is saved, but no memory is pointed yet by p. To reserve a memory that will be pointed by p, dynamically, we need an *allocation* operation. This operation is special to pointers. It is done using specific statements that allow to allocate dynamically the memory to be pointed (or refereed by the pointer). This allocation is done by anyone of the following statements:
  - ***p←new()***; //or it can be written also as: ***new(p);***

- p←***alloc()***; //or it can be written also as: ***alloc(p);***
- p←***malloc();*** //or it can be written also as: ***malloc(p);***

Any one of the above statements can be used to create a new address (so, to reserve a memory cell) to which p points. It is also possible to specify the size of the basic type when the allocation is done as follows: p←***alloc(sizeof(integer))***; which creates a pointer p that points on an integer. Idem for the *malloc* statement.

- **Access** to the pointed values: after the creation (or allocation), p points now to some value that can be accessed. The pointed value can be accessed using the following names: **(*p)**, $(p \rightarrow)$, or $(p\wedge)$. For example, the following algorithm declares a pointer p on integers, make an allocation, and accesses the content.

```
Var p: Pointer(integer);
Begin
    new(p); //create the pointer, allocating the memory to be pointed by p
    (*p)← 3;//access the pointed value and its initialization with value 3
    write(*p); //depict the value pointed by p
...
End.
```

Figure 4.1.3 shows how the previous algorithm affects the memory.

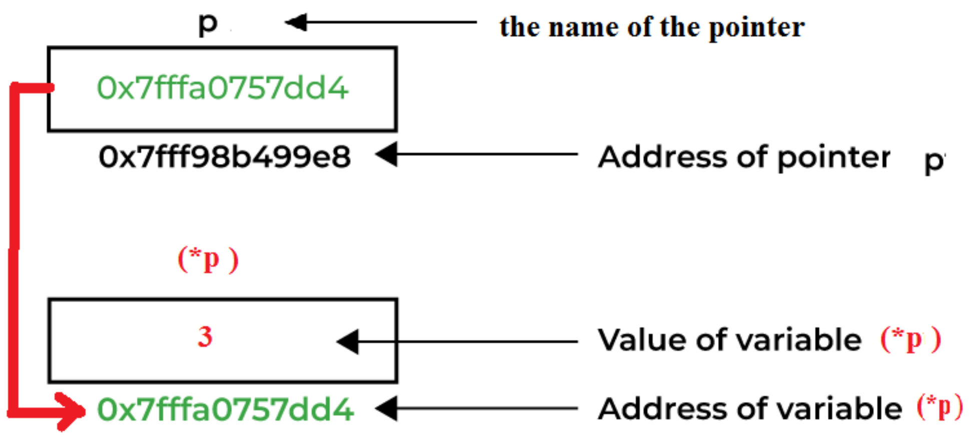


Figure 4.3: A pointer and the pointed value, creation and access.

- The release of the pointer: After the using of the pointer (through accessing, updating, etc), the programmer can at any time free the pointer (i.e., delete the pointer or deallocate the reserved memory) dynamically. This can be done using one of the following statements:

  - ***free(p);***
  - ***delete(p);***

  Executing a $free(p)$; on the previous pointer will give the following figure:

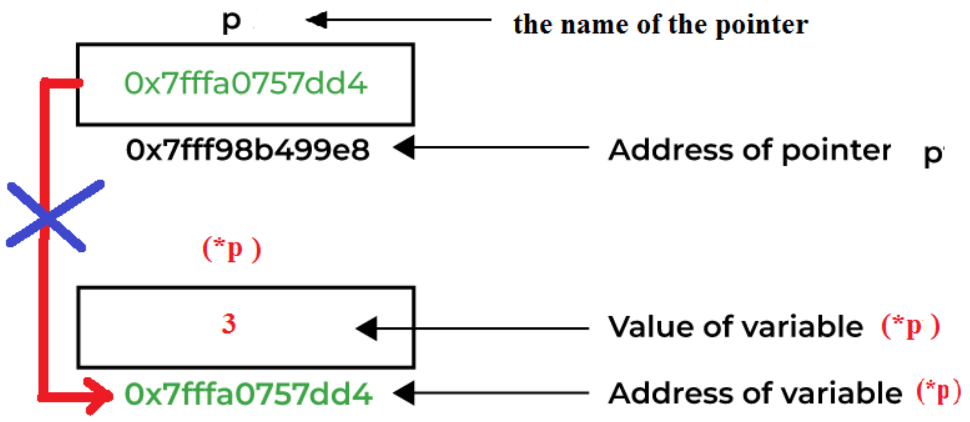


Figure 4.4: A deleted pointer, or a free pointer.

  In the above figure, the ***free*** operation on ***p*** causes the breaking of the link between ***p*** and ***(*p)***, which means that the cell (0x7fffa0757dd4) is no more reserved to the programmer.

- **Assignment of Nil constant**: *Nil* is a specific constant that can be assigned to any kind of pointers. Once **p** gets **Nil**, **p** does not point to any thing. In a correct programming style, the programmer has to assign the constant *Nil* to all released pointers.

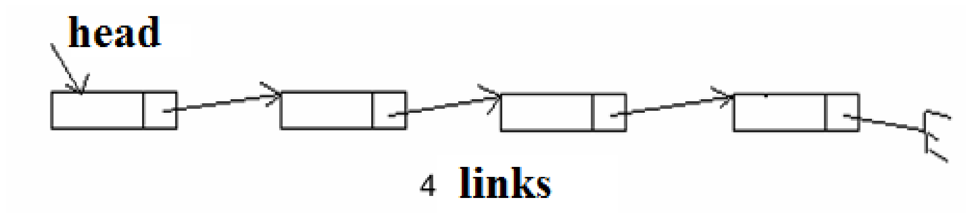
Now that we have presented the tools to manage dynamically the memory (by dynamic allocation and release), we can introduce the dynamic sequential structures in the next sections. This chapter will present: Linked Linear Lists (LLL), Linked Bidirectional Lists (LBL), Stacks, and Queues. These structures are also known as **Abstract Data Types** (ADT).

3. Definition (Abstract Data Type): An abstract data type (ADT) is a high level type which is defined independently of how it will be implemented. The word abstract means that the type is defined without focusing on how it will be actually programmed. So that, an ADT is characterized basically by a set of operations (called services) that are provided to its user. We say that an ADT has a specification (description) that describes the set of its operations in a high level way (usually in an algebraic way). The following sequential structure will be presented as ADTs before moving to their concrete implementation which can either static (using arrays) or dynamic (using Pointers).

4.2 Linked Linear Lists

4.2.1 Definition

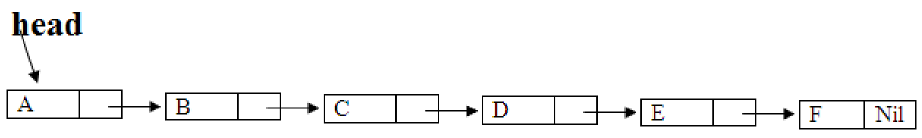
A linear linked list (LLC) or *"Linear linked lists"* is a set of links (or elements), dynamically allocated, linked together. Schematically, it can be represented as follows:



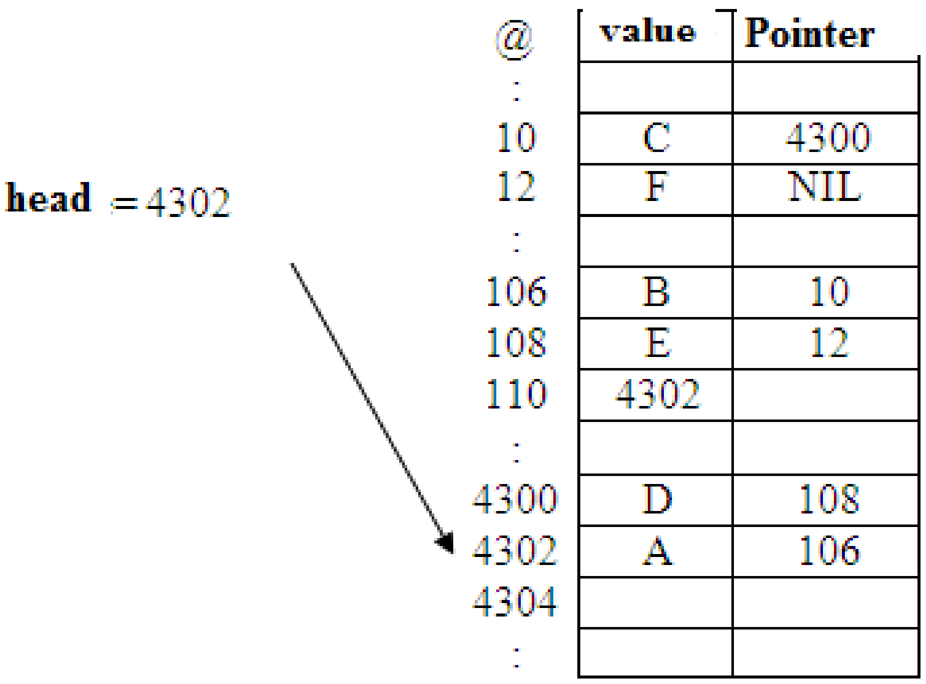
An element (or link) of an LLC is always a structure (compound object) with two fields: a *Value* field containing the information and an *Address* field giving the address of the next element. Each link is associated with an address.
A linear linked list is characterized by the address of its first element often called *head*. *Nil*, the address which does not point to any link, used to indicate the end of the list in the last link.

4.2.2 Representation

Consider the linear chained list represented by the following figure:



The actual in-memory representation of this list looks like the following:



@	value	Pointer
:		
10	C	4300
12	F	NIL
:		
106	B	10
108	E	12
110	4302	
:		
4300	D	108
4302	A	106
4304		
:		

4.2.3 Declaration

In algorithmic language, we will define the type of a link as follows:

```
Type  TElement = Record
    Value : Basic_Type; // any basic type
    Next : Pointer(TElement);
Fin;
```

Instead the keyword **record**, we can use **struct** (as in the C language).
Then we need a global variable to be defined as the head of the list as follows:

```
Var head:Pointer(TElement);
```

4.2.4 Operations

As explained before, A LLL is considered as an abstract data type defined by a set of operations, as follows:

```
Create: → LLL; // create an empty list
Insert: LLL x Basic_Type → LLL; // add a value to the list
Search: LLL x Basic_Type → Boolean; // search a value in the list
Delete: LLL x Basic_Type → LLL; // delete a value from the list
```

4.2.4.1 Implementation of the operations: A Dynamic implementation using Pointers

```
Procedure Create();
Begin
    Head←Nil;
End;
```

```
Procedure Insert(x:Basic_Type);
var p: Pointer(Basic_Type);
Begin
    //insertion of x at the head of the list
    alloc(p); (p->Value)←x; (p->Next)←Head; Head←p;
End;
```

The following algorithm finds an element x in the list.

```
Function Search(x:Basic_Type) : Boolean;
var p: Pointer(Basic_Type); exist: Boolean;
Begin
    exit←False;
    p←Head;
    While (p!=nil and not exist) Do
        If ((p->Value)=x) Then
            exist ← True;
        Else
            p←(p->Next);
        End If;
    End While;
    Search←exist;
End;
```

The following algorithm deletes an element x from the list. The element must be firstly founded then deleted. If there are several occurrences of x, we delete the first occurrence. If the element x does not exist then no thing is done. The implementation uses two pointers p which points the x if founded, and pr (i.e. previous) which points on p.

```
Procedure Delete(x:Basic_Type);
Var pre, p: Pointer(Basic_Type);
    Del: Boolean;
Begin
    pre←Head;
    p←Head;
    If (Head!=nil) Then
        If ((Head->Value)=x) Then
            Head←(Head->Next);
            Delete(p);
        Else
            Del←False;
            p←(Head->next);
            While (p!=nil and not Del) Do
                If ((p->Value)=x) Then
                    pre←(p->Next);
                    delete(p);
                    Del←True;
                Else
                    pre←p;
                    p←(p->Next);
                End If;
            End While;
        End If;
    End If;
End;
```

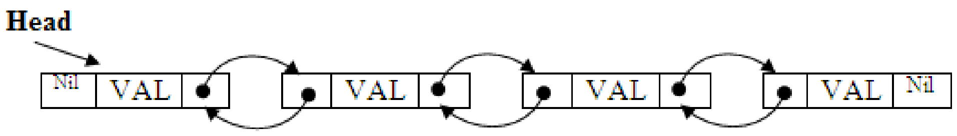
4.2.5 Other algorithms

- Browse algorithms: access by value, access by position,...
- Update algorithms: insertion, deletion,...
- Algorithms on several LLCs: merger, inter-classification, splitting,...

- Sorting algorithms on LLCs: bubble sort, merge sort,...

4.3 Linked Bidirectional Lists

It is an LLC where each link contains both the address of the previous element and the address of the next element which allows the list to be traversed in both directions.



Déclaration

```
Type TElement = Record
    Valeur : Basic_Type; // designates any type
    Prrevious, Next : Pointer(TElement);
End;
Var Head : Pointer(TElement);
```

4.3.1 Operations

As explained before, A BLL is considered as an abstract data type defined by a set of operations, as follows:

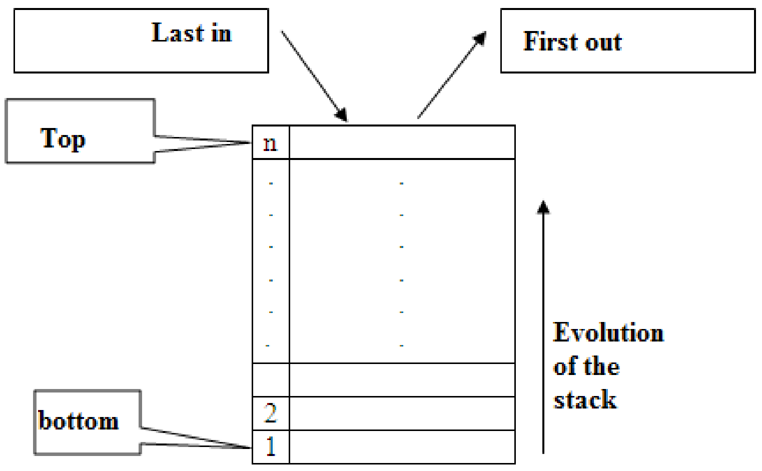
```
Create_BLL: → BLL; // create an empty list
Insert: BLL x Basic_Type → BLL; // add a value to the BL-list
Search: BLL x Basic_Type → Boolean; // search a value in the BL-list
Delete: BLL x Basic_Type → BLL; // delete a value from the BL-list
```

4.4 stacks

4.4.1 Definition

A stack is an ordered list of elements where insertions and deletions of elements occur at one end of the list called *the top of the stack.*

The principle of adding and removing from the stack is called LIFO (*Last In First Out*): "the last one in is the first one out"



4.4.2 Using of stacks

4.4.2.1 In a web browser

A stack is used to remember 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 "Show previous page" button.

4.4.2.2 Undoing operations

A word processor's "Undo Typing" function stores changes to text in a stack.

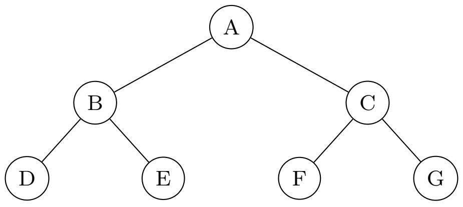
4.4.2.3 Call management in program execution

Fact(4)=4*Fact(3)=4*3*Fact(2)=4*3*2*Fact(1)=4*3*2*1=4*3*2=4*6=24

			1			
		2*Fact(1)	2*Fact(1)	2		
	3*Fact(2)	3*Fact(2)	3*Fact(2)	3*Fact(2)	6	
Fact(4)	4*Fact(3)	4*Fact(3)	4*Fact(3)	4*Fact(3)	4*Fact(3)	24

4.4.2.4 Depth traversal of trees

Consider the following tree:



The depth traversal algorithm is as follows:

```
Put Root in the stack;
While (The stack is not empty) Do
    Remove a node from the stack;
    Show its value;
    If (Node has children) Then
        Add these children to the stack
    End If;
End While;
```

The result will be: A, B, D, E, C, F, G.

4.4.2.5 Evaluation of postfix expressions

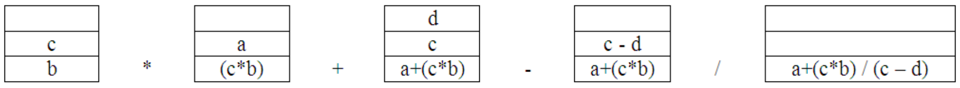
For the evaluation of arithmetic or logical expressions, programming languages generally use the prefix and postfix representation. In the postfix representation, we represent the expression by a new one, where the operations always come after the operands.

Example

The expression $((a + (b * c))/(c - d)$ is expressed, in postfix, as follows: $bc * a + cd - /$
To evaluate it, we use a battery. We traverse the expression from left to right, executing the following algorithm:

```
$i \leftarrow 1$;
**While** ($i < Length(Expression)$) **Do**
    **If** ($Expression[i]$ is an Operator) **Then**
        Remove two items from the stack;
        Calculate the result according to the operator;
        Put the result in the stack;
    **Else**
        Put the operand on the stack;
    **End If**;
    $i \leftarrow i + 1$;
**End While**;
```

The following diagram shows the evolution of the contents of the stack, by running this algorithm on the previous expression.



4.4.3 Operations on stacks

We consider that a stack of a set of elements (from a Basic_Type) is an abstract data type defined by a set of operations, as follows:

```
Create_Stack: → Stack; // create an empty stack
GetTop: Stack → Basic_Type; // returns the value in the top of the stack
Push: Stack x Basic_Type → Stack; // push a given value to the top of the stack
Pop: Stack → Stack; // delete the value from the top, the last added value
Empty: Stack → Boolean; // check the emptiness of the stack
Full: Stack → Boolean; // check if the stack is full, however this operations depends
only on the implementation of the stack
```

Exercise: Give the stack status after performing the following operations on an empty stack:
push(a), push(b), pop, push(c), push(d), pop, push(e), pop, pop.

4.4.4 Stack implementation

Stacks can be represented in two ways: by arrays or by LLLs:

4.4.4.1 Implementation using arrays

The static implementation of stacks uses arrays. In this case, the stack capacity is limited by the size of the array. Adding to the stack is done in the ascending direction of the indices, while removal is done in the opposite direction.

4.4.4.2 Implementation by LLLs

The dynamic implementation uses linear lists. In this case, the stack can be empty, but can never be full, except of course in the event of insufficient memory space. Stacking and unstacking in dynamic stacks is at the top of the list.

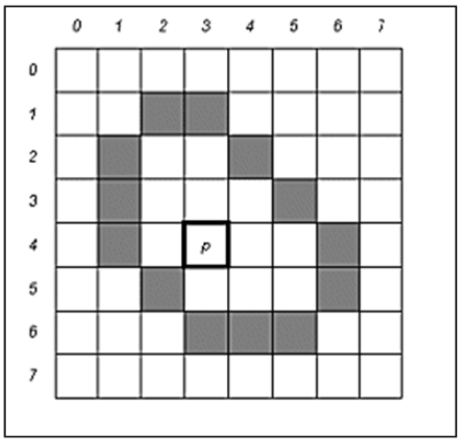
The following two algorithms "StackAsArray" and "StackAsLLL" present two examples of static and dynamic implementation of stacks of integers.

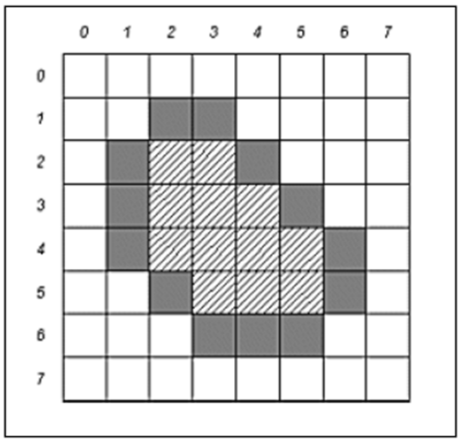
```
Algorithm StackAsArray;
Var  Stack : Array[1..n] of integer;      Top : integer;
Procedure Create_Stack();
Begin
    $Top \leftarrow 0$ ;
End;
Function Empty() : Boolean;
Begin
    $Empty \leftarrow (top = 0)$ ;
End;
Function Full() : Boolean;
Begin
    $Full \leftarrow (Top = n)$ ;
End;
Procedure Push( x : integer);
Begin
    If (Full) Then
        Write('Unable to stack, the stack is full!!')
    Else
        $top \leftarrow top + 1$ ;
        $Stack[Top] \leftarrow x$ ;
    End If;
End;
Procedure Pop();
Begin
    If (Empty) Then
        Write('Cannot pop, the stack is empty!!')
    Else
        $Top \leftarrow Top - 1$ ;
    End If;
End;

Begin

End.
... Using the stack ...
```

```
Algorithm StackasLLLs;
Type TElement = Record
    Value : integer;
    Next : Pointer(TElement);
End;
Var P, Top : Pointer(TElement);
Procedure Create_Stack();
Begin
    $Top \leftarrow Nil$ ;
End;
Function Empty() : Booleen;
Begin
    $Empty \leftarrow (Top = Nil)$ ;
End;
Function GetTop() : integer;
Begin
    If (not Empty) Then
        $GetTop \leftarrow (Top- > Value)$ ;
    End If;
End;
Procedure Push( x : integer);
Begin
    Alloc(P);          (P->Value)$\leftarrow$ x;
    (P->Next)$\leftarrow$ Top;       Top $\leftarrow$ P;
End;
Procedure Pop();
Begin
    If (Empty) Then
        Write('Impossible to pop, the stack is empty!!')
    Else
        P $\leftarrow$ Top;       Top $\leftarrow$ (Top->Next);       Delete(P);
    End If;
End;

Begin
    ... Using the stack in the main program ...
End.
```

4.4.5 Application example: Filling an area of an image

An image in computing can be represented by a matrix of points $'Image'$ having M columns and N rows. An element $Image[x, y]$ of the matrix represents the color of the point p with coordinates (x, y). We propose to write here a function which, from a point p, *spreads* a color c around this point. The progression of the spread color stops when it encounters a color other than that of point p. The following figure illustrates this example, considering $p = (3, 4)$.





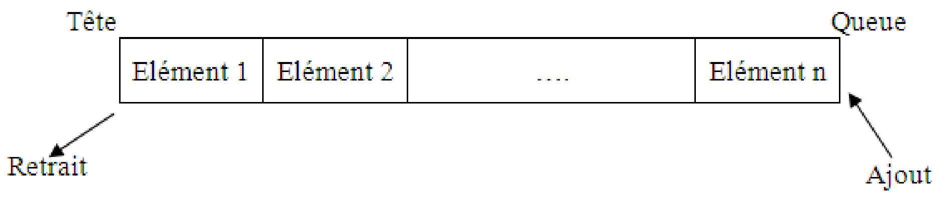
To fill, we must go in all directions from point p. This resembles the path of a tree with the nodes of four threads. The following procedure resolves the issue using a stack.

```
**Procedure** Fill( $Image$ : $Array[0..M-1, 0..N-1]$ **of color**; $x, y$ : **integer**; $c$ : **color**);
**Var** $c_1$ : **color**;
**Begin**
    $c_1 \leftarrow Image[x, y]$;
    Create;
    Push($(x, y)$);
    **While** ($\neg$ Empty) **Do**
        (x,y)$\leftarrow$ Get_Top;
        Pop;
        **If** ($Image[x, y] = c_1$) **Then**
            $Image[x, y] \leftarrow c$;
            **If** ($x > 0$) **Then**
                Pus($(x-1, y)$)
            **End If**;
            **If** ($x < M-1$) **Then**
                Push($(x+1, y)$)
            **End If**;
            **If** ($y > 0$) **Then**
                Push($(x, y-1)$)
            **End If**;
            **If** ($y < N-1$) **Then**
                Push($(x, y+1)$)
            **End If**;
        **End If**;
    **End While**;
**End**;
```

4.5 Queues

4.5.1 Definition

The queue is a structure that allows objects to be stored in a given order and removed in the same order, i.e. according to the FIFO protocol *'first in first out'*. We always add an element at the bottom of the list and remove the one at the top.



4.5.2 Operations

As explained before, a queue is considered as an abstract data type defined by a set of operations, as follows:

Create_Queue: → Queue; // create an empty queue
Enqueue: Queue x Basic_Type → Queue; // add a value to the queue in the head
Dequeue: Queue x Basic_Type → Queue; // delete the value in the tail of the queue
GetHead: Queue → Basic_Type; //returns the value saved in the head of the queue
GetTail: Queue → Basic_Type; //returns the value saved in the tail of the queue
Empty: Queue → Boolean; // test if the queue is empty
Full: Queue → Boolean; // test if the queue is full, indeed this operation is optional and it depends really on the implementation

4.5.3 Using queues

Queues are used, in programming, to manage objects that are waiting for further processing, such as the management of documents to be printed, programs to be executed, messages received, etc. They are also used in tree traversals.

Exercise

Resume traversing the tree from section 4.4.2.4 (deep traversal) using a queue instead of the stack.

4.5.4 Implementing Queues

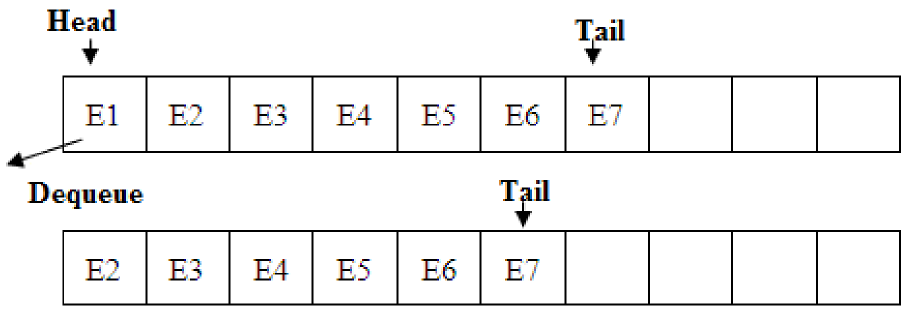
Similar to stacks, queues can be represented in two ways:

- by static representation using tables,
- by dynamic representation using linear linked lists.

4.5.4.1 Static implementation

The static implementation can be done by shifting using an array with a fixed head, always at 1, and a variable tail. It can also be done by flow using a circular array where the head and tail are both variable.

1. By **shift**



$\rightarrow$ The Head is fixed at the value 1 forever

$\rightarrow$ The Tail moves from 0 to n. It can increase or decrease, during the execution.

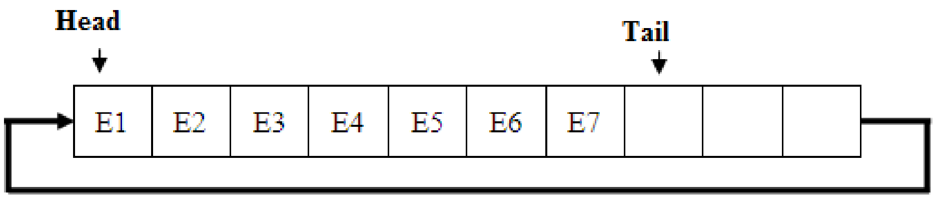
$\rightarrow$ The Tail refers (i.e., is an the index) to the last added element in the queue if the queue is not empty

$\rightarrow$ The queue is empty if $Tail = 0$

$\rightarrow$ Queue is full if $Tail = n$

$\downarrow$ shift issue on each scroll (i.e., dequeuing)

2. **By flow**: The queue is represented by a circular table



→ The Head and the Queue moves from 1 to n.

→ The Tail refers to en empty (unused cell) all the time. Thus, the last added element is not in the cell indexed by Tail but it is the previous cell to the cell indexed by Tail

→ The queue is empty if $Head = Tail$

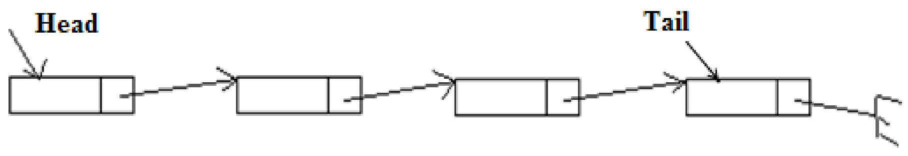
→ The queue is full if $(Tail + 1)\ mod\ n = Head$

↓ We sacrifice a cell to distinguish the case of an empty queue from that of a full queue.

Exercise Think of a solution that avoids sacrificing a square.

4.5.4.2 Dynamic Implementation

The dynamic representation uses a linear linked list. Enqueuing is done at the head of the list and Dequeuing is done at the tail. The queue, in this case, may become empty, but will never be full.



The two algorithms QueueByFlow and QueueByLLL, at the end of this section, present examples of implementation, respectively, static and dynamic.

4.5.5 Special queue (Priority queue)

A priority queue is a collection of items into which insertion does not always occur at the queue. Any new element is inserted into the queue according to its priority. Dequeuing is always done from the beginning.
In a priority queue, a priority item will take the Head in the queue even if it arrives last. An element is always accompanied by information indicating its priority in the queue.
The implementation of these queues can be an array or a LL-list, but the most efficient and widely used implementation uses special trees called 'heaps' (To be presented in chapter 5).

```
Var  Queue : Array[1..n] of integer;      Head, Tail : integer;
Procedure Create_Queue();
Begin
    $Head \leftarrow 1$; $Tail \leftarrow 1$;
End;
Function Empty() : Boolean;
Begin
    $Empty \leftarrow (Head = Tail)$ ;
End;
Function Full() : Boolean;
Begin
    $Full \leftarrow (((Tail + 1)\ mod\ n) = Head)$ ;
End;

Function Full() : Boolean;
Begin
    $Full \leftarrow (((Tail + 1)\ mod\ n) = Head)$ ;
End;

Procedure Deueue( );
Begin
    If (Empty) Then
        Write('Impossible to dequeue, queue is empty!!')
    Else
        $Head \leftarrow (Head + 1)\ mod\ n$ ;
    End If;
End;
```

```
Procedure Enqueue( x : integer);
Begin
    If (Full) Then
        Write('Impossible to enqueue, the queue is full!!')
    Else
        $Queue[Tail] \leftarrow x$ ;
        $Tail \leftarrow (Tail + 1)\ mod\ n$ ;
    End If;
End;

Function GetHead() : integer;
Begin
    If (not Empty) Then
        $GetHead \leftarrow Queue[Head]$ ;
    End If;
End;
Function GetTail() : integer;
Begin
    If (not Empty) Then
        If (Tail=1) Then
            $GetTail \leftarrow Queue[n]$
        Else
            $GetTail \leftarrow Queue[Tail - 1]$ ;
        End If;
    End If;
End;

... Using the queue ...
```

Algorithm 2: QueueByFlow

```
Type TElement = Record
    Value : integer;
    Next : Pointer(TElement);
End;
Var P, Head, Tail : Pointer(TElement);
Procedure Create_Queue();
Begin
    $Head \leftarrow Nil$ ;    $Tail \leftarrow Nil$ ;
End;
Function Empty_Queue() : Booleen;
Begin
    $Empty\_Queue \leftarrow (Head = Nil)$ ;
End;
Procedure Enqueue( x : integer);
Begin
    Alloc(P);    (P->Value)←x;    (P->Next)←Nil;
    If (Tail = Nil) Then
        $Head \leftarrow P$ ; // case of an empty queue
    Else
        $(Tail- > Next) \leftarrow P$;
    End If;
    $Tail \leftarrow P$ ;
End;
Procedure Dequeue();
Begin
    If (Empty_Stack) Then
        Write('Cannot delete any element, the queue is empty!!')
    Else
        $P \leftarrow Head$;
        $Head \leftarrow (Head- > Next)$;
        Delete(P); //free the cell
    End If;
End;
... Using the queue in the main program later...
```

```
Function GetHead() : integer;
Begin
    If (not Empty) Then
        $GetHead \leftarrow (Head- > Value)$ ;
    End If;
End;
Function GetTail() : integer;
Begin
    If (not Empty) Then
        $GetTail \leftarrow (Tail- > Value)$
    End If;
End;

... Using the queue in the main program later...
```

Algorithm 3: QueueByLLL