University Mohamed Khider-Biskra
Computer Science Department

1^{st} year Computer Science Degree — Chapter 3
ASD2 — 2023/2024

Linked lists

1 Linear Data Structures

Among the linear data structures there are:

- Arrays,
- Linked lists,

Linear data structures induce a notion of sequence between the component elements (1st, 2nd, 3rd, next, last $\cdots$).

1.1 Arrays

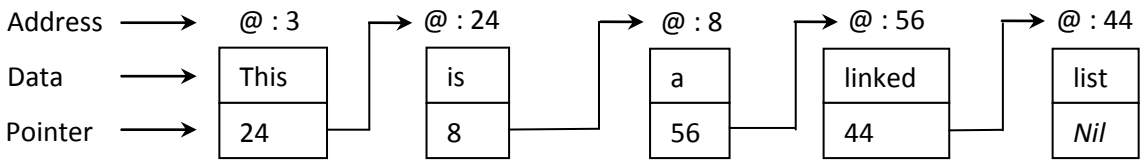
You already know the linear structure of the array type for which component elements of the same type are placed contiguously in memory.
To create an array, with 1 or 2 dimensions, you must know its size which cannot be modified during the program execution, and associate it with an index to browse its elements. For arrays, the sequence corresponds to the numbers of the cells in the array. We access directly to an element of the array through its index.
Consider the following 1-dimensional array named T:

12	14	10	24

To reach the third cell of the array it is enough to write T[3] which contains 10, if the values of the index start at 1.
The array-like structure poses problems for inserting or deleting an element because these actions require shifting the contents of the array cells which take time in the execution of a program.
This type of value storage can therefore be expensive in execution time. There is another structure, called a linked list, to store values, this structure makes it easier to insert and delete values in a linear list of elements.

1.2 Linked lists

A linked list is a linear structure that has no fixed dimension when it is created. Their Elements of the same type are scattered in memory and linked together by pointers. Its dimension can be modified depending on the space available in memory. The list is accessible only by its head, that is to say its first element.
For linked lists, the sequence is implemented by the pointer carried by each element which indicates the location of the next element. The last element in the list points to nothing (*Nil*). We access an element of the list by traversing the elements using their pointers.
Consider the following linked list (@ indicates that the number following it represents an address):



To access the third element of the list you must always start reading the list with its first element in whose pointer the position of the second element is indicated. In the pointer of the second element of the list we find the position of the third element?
To add, delete or move an element, it suffices simply to allocate a place in memory and updates the element pointers.
There are different types of linked lists:

- Simple linked list consisting of elements linked together by pointers.
- Ordered linked list where the next element is greater than the previous one. Insertion and deletion of elements is done so that the list remains sorted.
- Doubly linked list where each element has not only one but two pointers pointing to the previous element and the next element respectively. This allows you to read the list in both directions, from the first to the last element or vice versa.
- Circular list where the last element points to the first element in the list. If it is a doubly linked list then first element also points to the last.

These different types can be mixed according to needs.
We use a linked list rather than an array when we need to process objects represented by sequences on which we must make numerous deletions and numerous additions. The manipulations are then faster than with arrays.
Summary

Structure	**Dimension**	**Position of information**	**Access to information**
Array	Fixed	By its index	Directly by the index
Linked list	Evolves according to the actions	By its address	Sequentially by the pointer of each element

2 Linked lists

2.1 Definitions

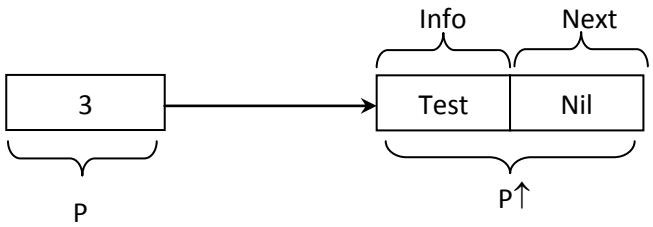
An element of a list is the set (or structure) formed by :

- a data or information,
- a pointer named Next indicating the position of the next element in the list.

Each element is associated with a memory address.
Linked lists use the concept of dynamic variables.
A dynamic variable:

- is declared at the start of the execution of a program,
- it is created there, that is to say, a space is allocated to it to occupy at an address in the memory,
- it can be destroyed there, that is to say the memory space it occupied is freed,
- access to the value is done using a pointer.

A pointer is a variable whose value is a memory address. A pointer, denoted P, points to a dynamic variable denoted $P\uparrow$.
The base type is the type of the pointed variable.
The pointer type is the set of addresses of the pointed variables of the base type. It is represented by the symbol $\uparrow$ followed by the base type identifier.
Example



The pointer variable P points to the memory space $P\uparrow$ with address 3. This memory cell contains the value "Test" in the *Info* field and the special value *Nil* in the *Next* field. This field will be used to indicate what the next element is when the cell will be part of a list. The Nil value indicates that there is no next item. $P\uparrow$ is the object whose address is stored in P.
Linked lists result in the use of procedures of dynamic allocation and release of memory. These procedures are as follows:

- $Allocate(P)$ or $New(P)$: reserves a memory space $P\uparrow$ and gives the address of this memory space as value to P. We allocate memory space for an element pointed to by P.

- $Deallocate(P)$ or $Free(P)$: frees the memory space which was occupied by the element to be deleted $P\uparrow$ on which P points to.

To define the variables used in the example above, you must:

- define the type of list of elements:

  Type $Cell$ = **Record**

  $Info$: String;

  $Next$: $List$;

  End;

- define the pointer: **Type** $List = \uparrow Cell$;
- declare a pointer variable: **Var** P : $List$;
- allocate a memory cell which reserves space in memory and gives P the value of the address of the memory space $P\uparrow$: $Allocate(P)$;
- assign values to the memory space $P\uparrow$:

  $$P\uparrow .Info \longleftarrow \text{“Test”}; \qquad P\uparrow .Next \longleftarrow Nil;$$

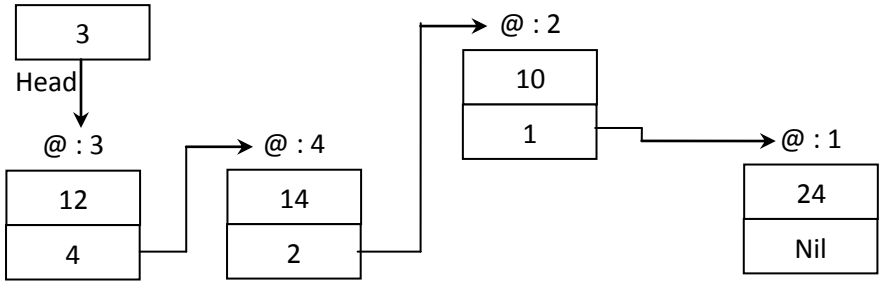
When $P = Nil$ then P points to nothing.

2.2 Simple linked lists

A simple linked list is composed of:

- a set of elements such that each:
  - is stored in memory at a certain address,
  - contains data ($Info$),
  - contains a pointer, often named $Next$, which contains the address of the next element in the list,
- a variable, called $Head$, containing the address of the first element of the linked list.

The last element pointer contains the value Nil. In the case of an empty list the pointer of the header contains the value Nil. A list is defined by the address of its first element.
Before writing algorithms manipulating a linked list, it is useful to show graphically a diagram representing the organization of the elements of the linked list.
Example Let's consider the following list of integers: 12, 14, 10, 24. The corresponding linked list could be:
The 1^{st} element of the list is 12 at address 3 (start of the linked list)



The 2^{nd} element of the list is 14 at address 4 (because the pointer of the cell at address 3 is equal to 4)
The 3^{rd} element of the list is 10 at address 2 (because the pointer of the cell at address 4 is equal to 2)
The 4^{th} element of the list is 24 at address 1 (because the pointer of the cell at address 2 is equal to 1)

If P has the value 3	If P has the value 2
$P\uparrow.Info$ has a value of 12	$P\uparrow.Info$ has a value of 10
$P\uparrow.Next$ has a value of 4	$P\uparrow.Next$ has a value of 1

2.3 Basic processing of the use of a simple linked list

You must start by defining a type of variable for each element of the list. In algorithmic, this is done as follows:

Type

$List = \uparrow Element$;
$Element =$Record
$Info$: string;
$Next : List$;
End;

Var $Head, P : List$;

The type of $Info$ depends on the values contained in the list: integer, character string, varying for some type...
Generally, treatments of lists are as follows:

- Creating a list;
- Inserting an element;
- Deleting an element;
- Modifying an element;
- Traversing a list;
- Searching for a given value in a list.

2.3.1 Creating a linked list composed of two elements of string type

Declaration
Type

$List =\uparrow Element$;
$Element =$Record
$Info$: string;
$Next : List$;
End;

Algorithm Create List of Two Elements;
Var

$Head, p : List$;

Begin

$Head \leftarrow Nil$; /*for the moment, the list is empty*/
New(p); /*allocate a memory space for the first element*/
Read($p \uparrow .Info$); /*save in the field Info of the element pointed by p the entered value*/
$p \uparrow .Next \leftarrow Nil$; /*there is not a next element*/
$Head \leftarrow p$; /*Head points to $p \uparrow$*/
/*We must now add the 2^{nd} element; i.e, insert an element at the head of the list */
New(p); /*allocate a memory space for the second element*/
Read($p \uparrow .Info$); /*save in the field Info of the element pointed by p the entered value*/
$p \uparrow .Next \leftarrow Head$; /*the element is inserted in the head of the list*/
$Head \leftarrow p$;

End.

2.3.2 Create a linked list composed of several string elements

Type declarations for the list:
Type

$List =\uparrow Element$;
$Element =$ Record
$Info$: string;
$Next : List$;
End;

To create a linked list containing a number of elements to be specified by the user, it suffice to introduce two variables $NumberElt$ (Number of Elements) and $Counter$ of type Integer.

- Allow the user to enter the value of $NumberElt$ at the beginning,
- Write a loop for $Counter$ going from 1 to $NumberElt$ including the above four instructions.

Algorithm Create List of Known Number of Elements;
Var
$Head, p : List$;
$NumberElt, Counter$: Integer;
Begin
Read($NumberElt$);
$Head \leftarrow Nil$;
For $Counter \leftarrow 1$ to $NumberElt$ **do**
New(p); /*allocate a memory space for the element to be added*/
Read($p \uparrow .Info$); /*save in the field $Info$ of the element pointed by p the entered value*/
$p \uparrow .Next \leftarrow Head$; /*the element is inserted in the head of the list*/
$Head \leftarrow p$; /*The pointer $Head$ points now on p*/
End For
End.

To create a linked list containing an unknown number of elements you must:

- declare a reading variable of the same type as that of the information carried by the list,
- determine and indicate to the user the value he must enter to announce that there is no longer element to add in the list (for example, "XXX"),
- write a While loop to execute the above four instructions as long as the value entered by the user is different from the value indicating the end of elements addition.

Algorithm Create List of Unknown Number of Elements;
Var
$Head, p : List$;
$Resp$: String;
Begin
$Head \leftarrow Nil$;
Read($Resp$);
While $Resp \neq$ "XXX" **do**
New(p); /*allocate a memory space for the element to be added*/
$p \uparrow .Info \leftarrow Resp$; /*save in the field $Info$ of the element pointed by p the entered value*/
$p \uparrow .Next \leftarrow Head$; /*the element is inserted in the head of the list*/
$Head \leftarrow p$; /*The pointer $Head$ points now on p*/
Read($Resp$);
End While
End.

2.3.3 Display elements of a linked list

A simple linked list can only be traversed from the first to the last element.
The algorithm is given as a procedure which receives the head of the list as a parameter.

Procedure Display List ($p : List$);
Begin
$p \leftarrow Head$; /*p points on the 1st element of the list*/
While $p \neq Nil$ **do** /*We traverse the list as long as the address of the next element is not Nil*/
Write($p \uparrow .Info$); /*display the value contained at the address pointed by p*/
$p \leftarrow p \uparrow .Next$; /*We move on to the next element*/
End While
End;

2.3.4 Find a given value in an ordered linked list

In this example we take the case of a linked list containing elements of type string, but it could be any other type, depending on the one determined when the list was created (remember that all elements of a linked list must have the same type). The list will be traversed from its first element (the one pointed to by the head pointer). He has two termination cases:

- having found the value of the element,
- have reached the end of the list.

The algorithm is given in the form of a procedure which receives the head of the list as a parameter and the searched value.

Procedure SearchValueList ($Head : List, Val$: String);
Var
  $p : List$;
  $Find$: Boolean;
Begin
  If $Head \neq Nil$ **then**
    $p \leftarrow Head$;
    $Find \leftarrow$ False;
    While $p \neq Nil$ and Not($Find$) **do**
      If $p \uparrow .Info = Val$ **Then**
        $Find \leftarrow$ True;
      Else
        $p \leftarrow p \uparrow .Next$;
      End If
    End While
    If $Find$ **Then**
      Write ("The Value", Val," is in the list");
    Else
      Write ("The Value", Val," is not in the list");
    End If
  Else
    Write ("The list is empty");
  End If
End;

2.3.5 Manipulation in C language

Before using dynamic memory allocation, you should include the library stdlib.h

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

Algorithm	**C Program**
Var p :↑basic_type	basic_type* p;
New(p);	p =(castType*) malloc(size);
free(p);	free(p);

University Mohamed Khider-Biskra
Computer Science Department

1^{st} year Computer Science Degree | Chapter 4
ASD2 | 2023/2024

Abstract Data Types: Stacks and Queues

1 Abstract vs Concrete data type

In computer science, an abstract data type (ADT) is a mathematical model for data types, defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This mathematical model contrasts with data structures or concrete data types, which are concrete representations of data, and are the point of view of an implementer, not a user.

ADTs are a theoretical concept, used in formal semantics and program verification and, less strictly, in the design and analysis of algorithms, data structures, and software systems. Most mainstream computer languages do not directly support formally specifying ADTs. However, various language features correspond to certain aspects of implementing ADTs, and are easily confused with ADTs proper; these include abstract types, opaque data types, protocols, and design by contract. For example, in modular programming, the module declares procedures that correspond to the ADT operations, often with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs, but the module only informally defines an ADT. The notion of abstract data types is related to the concept of data abstraction.

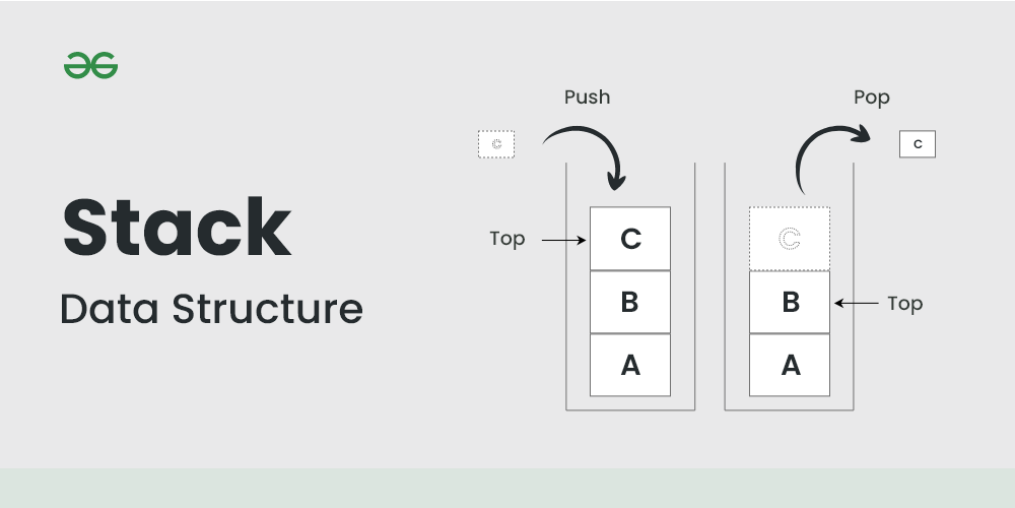
In our course, we are mainly interested with two ADTs: Stacks and Queues.

2 Stacks

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.

What is Stack Data Structure?

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It behaves like a stack of plates, where the last plate added is the first one to be removed.



2.1 Basic Operations of Stack Data Structures

- Push: Adds an element to the top of the stack.
- Pop: Removes the top element from the stack.
- Peek (or Top): Returns the top element without removing it.
- IsEmpty: Checks if the stack is empty.
- IsFull: Checks if the stack is full (in case of fixed-size arrays).

2.2 Applications of Stack Data Structures

- Recursion
- Expression Evaluation and Parsing
- Depth-First Search (DFS)
- Undo/Redo Operations
- Browser History

2.3 Implementation

Stacks may be implemented in two different manners: by arrays or by linked lists.

2.3.1 Implementation by arrays

The static implementation of stacks uses arrays. In this case, the number of elements that the stack may contains (its 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.

2.3.2 Implementation by linked lists

The dynamic implementation uses linear lists. In this case, the stack can be empty, but can never be full, except of course in the case of insufficient memory space. Adding and removing elements in dynamic stacks is done at the head of the list.
The following two algorithms present static and dynamic implementation examples of stacks.

```
Algorithm Static_Stack;
Const   N = 20;          /*for example, the capacity is 20 element*/
Type    Stack = Record
                  S: Array[N] of element_type;
                  Top : Integer;
                End;
Var   p : Stack;
Procedure InitStack(var p : Stack);
Begin
    p.Top ← 0;
End;
Function Is_empty(p : Stack): Boolean;
Begin
    Is_empty ← (p.Top = 0);
End;
Function Is_full(p : Stack): Boolean;
Begin
    Is_full ← (p.Top = N);
End;
Procedure Push(x : type_element, var p : Stack);
Begin
    If Is_full(p) Then
       Write("The stack is full");
    Else
       p.Top ← p.Top + 1;
       p.S[p.Top] ← x;
    EndIf
End;
Procedure Pop(var x : type_element, var p : Stack);
Begin
    If Is_empty(p) Then
       Write("The stack is empty");
    Else
       x ← p.S[p.Top];
       p.Top ← p.Top − 1;
    EndIf
End;
Begin
    ··· Use of the stack ···
End.
```

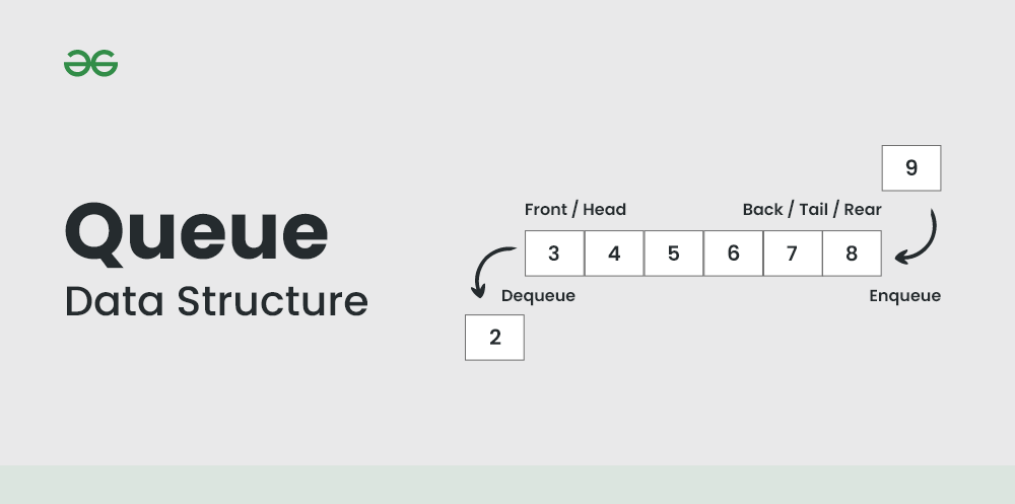
```
Algorithm Dynamic_Stack;
Type    Element = Record
                    Val: type_element;
                    next :↑ Element;
                  End;
        Stack :↑ Element;
Procedure InitStack(p : Stack);
Begin
        p ← Nil;
End;
Function Is_empty(p : Stack): Boolean;
Begin
        Is_empty ← (p = Nil);
End;
Procedure Push(x : type_element, var p : Stack);
Var l :↑ Element;
Begin
        New(l);
        l ↑ .Val ← x;
        l ↑ .next ← p;
        p ← l;
End;
Procedure Pop(var x : type_element, var p : Stack);
Var l :↑ Element;
Begin
        If Is_empty(p) Then
           Write("The stack is empty");
        Else
           x ← p ↑ .Val;
           l ← p;
           p ← p ↑ .next;
           free(l);
        EndIf
End;
Begin
        ··· Use of the stack ···
End.
```

3 Queues

A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of ?First in, First out? (FIFO), where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

What is Queue in Data Structures?

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It operates like a line where elements are added at one end (rear) and removed from the other end (front).



3.1 Basic Operations of Queue Data Structure

- Enqueue (Insert): Adds an element to the rear (or tail) of the queue.
- Dequeue (Delete): Removes and returns the element from the front (head) of the queue.
- Peek (or First): Returns the element at the front of the queue without removing it.
- isEmpty: Checks if the queue is empty.
- isFull: Checks if the queue is full.

3.2 Applications of Queue

- Task scheduling in operating systems
- Data transfer in network communication
- Simulation of real-world systems (e.g., waiting lines)
- Priority queues for event processing queues for event processing

3.3 Implementation

As for stacks, queues may be also represented either statically by arrays or dynamically by linked lists.

3.3.1 Implementation by arrays

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 carried out by flow using a circular array where the head and tail are both variable.

1. By shifting
    - the queue is empty if $tail = 0$.
    - the queue is full if $tail = N$.
    - shifitng problem at each Dequeue.
2. By flow: The queue is represented by a circular array.
    - the queue is empty if $head = tail$.
    - the queue is full if $(tail + 1) \bmod N = head$.
    - We sacrifice a cell to distinguish the case of an empty queue from that of a full one.

3.3.2 Implementation by linked lists

The dynamic representation uses a linear linked list. Enqueue is done at the head of the list and dequeue moves to the tail. The queue, in this case, can become empty, but it will never be full.
The two following algorithms present examples of static and dynamic implementations of queues.

```
Algorithm Static_Queue_by_flow;
Const   N = 20;          /*for example, the capacity is 20 element*/
Type    queue = Record
                  T: Array[N] of element_type;
                  head, tail : Integer;
                End;
Var   q : queue;
Procedure InitQueue(var q : queue);
Begin
    q.head ← 1; q.tail ← 1;
End;
Function Is_empty(q : queue): Boolean;
Begin
    Is_empty ← (q.head = q.tail);
End;
Function Is_full(q : queue): Boolean;
Begin
    Is_full ← ((q.tail + 1) mod N = q.head);
End;
Function Peek(q : queue): element_type;
Begin
    If Is_empty(q) Then
       Write("The queue is empty");
    Else
       Peek ← q.T[q.head];
    EndIf
End;
Procedure Enqueue(x : element_type, var q : queue);
Begin
    If Is_full(q) Then
       Write("The queue is full");
    Else
       q.T[q.tail] ← x;
       q.tail ← (q.tail + 1) mod N;
    EndIf
End;
Procedure Dequeue(var x : element_type, var q : queue);
Begin
    If Is_empty(q) Then
       Write("The queue is empty");
    Else
```

$x \leftarrow p.T[q.head]$;
$q.head \leftarrow (q.head + 1) \bmod N$;
EndIf
End;
Begin
$\cdots$ Use of the Queue $\cdots$
End.

```
Algorithm Dynamic_Queue;
Type    Element = Record
                    Val: element_type;
                    next :↑ Element;
                  End;
        queue = Record
                    head, tail :↑ Element;
                  End;
Procedure InitQueue(var q : queue);
Begin
    q.head ← Nil; q.tail ← Nil;
End;
Function Is_empty(q : queue): Boolean;
Begin
    Is_empty ← (q.head = Nil);
End;
Function Peek(q : queue): element_type;
Begin
    If Is_empty(q) Then
       Write("The queue is empty");
    Else
       Peek ← q.head ↑ .Val;
    EndIf
End;
Procedure Enqueue(x : element_type, var q : queue);
Var l :↑ Element;
Begin
    New(l);
    l ↑ .Val ← x;
    l ↑ .next ← Nil;
    If q.tail = Nil Then
       q.head ← l;
    Else
       q.tail ↑ .next ← l
    EndIf
    q.tail ← l;
End;
Procedure Dequeue(var x : element_type, var q : queue);
var l :↑ Element;
Begin
    If Is_empty(p) Then
```

```
        Write("The queue is empty");
    Else
        x ← q.head ↑ .Val;
        l ← q.head;
        q.head ← q.head ↑ .next;
        free(l);
    EndIf
End;
Begin
    ··· Use of the Queue ···
End.
```