Chapter 8 : Pointers and Linked Lists

1. The pointers

Pointers play a fundamental role in many algorithms and programming languages, particularly in C. In algorithmics, pointers allow to access and manipulate data efficiently by referencing their memory address instead of direct values.

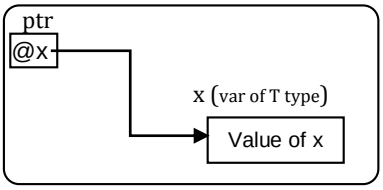
A pointer is a variable that contains the memory address of another variable. Therefore, when we handle a pointer in C, we have direct control over the computer memory. This provides considerable flexibility and power in designing efficient algorithms. Using pointers, programmers can dynamically allocate and free up memory, pass data addresses between functions, and implement complex data structures such as chained lists and trees.

1.1.Definition

A pointer is a variable that stores the memory address of another variable. It allows indirect access to the value stored at that address. A pointer is defined by its :

- name,
- pointed type and
- memory location reserved for a specific address (Value).

A pointer to a type T is a variable that contains the address of a variable of type T.



1.2. Elementary operators

- **Address-of Operator (@ or &):** Retrieves the memory address of a variable.
- **Indirection Operator (*):** Accesses the value stored at the memory address.

1.3. Pointer operations

1.3.1. Dynamic memory allocation and deallocation :

- Dynamic memory allocation is a crucial function used for the dynamic reservation of memory and returns of its address (pointer).
- The dynamic memory deallocation : when a dynamic memory location is no longer needed, it should be deallocated (freed) to release the reserved memory.

Syntax:

In Algorithmics	In C language
Allocate(ptr); // Dynamic memory allocation	ptr = **malloc**(**sizeof**(**int**));
free (ptr**);** // Dynamic memory deallocation	**free (**ptr**) ;**

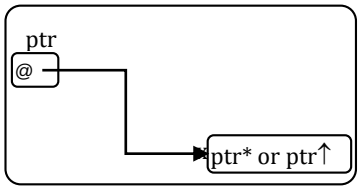
Note: After freeing memory, it is recommended to reset the pointer to nil.

1.3.2. Access to the pointed value

The operation to access the value pointed by a pointer, often called dereference, consists in retrieving the value stored at the memory address indicated by the pointer. In other words, it retrieves the actual data to which the pointer refers. This is done by using the dereference operator, usually represented by the "*" or "↑" symbol that follows the pointer's name.

For example, if we have a pointer named ptr that points to a variable x, to access the value of x through the pointer ptr, we use the dereference operation as follows: ptr*. (or *ptr in C). This operation enables the program to retrieve the actual value stored at the memory address referenced by the pointer, which is useful for accessing and manipulating data indirectly in algorithms.

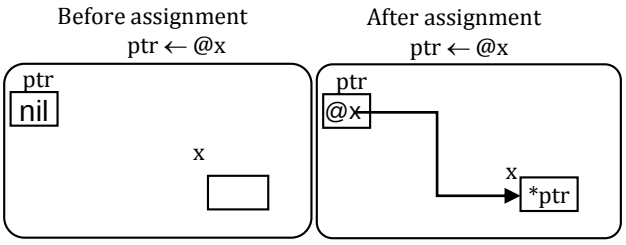
In Algorithmics	In C language
ptr* or ptr↑	*ptr



Notes :
- The pointer must contain a valid address.
- The address of a variable "x" is retrieved by : @x (In Algorithmics) and &x (In C language)

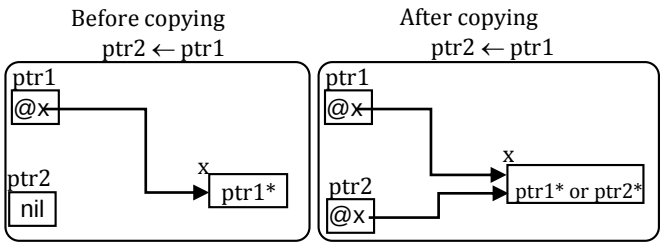
1.3.3. Initializing and assignment of a pointer

- Initializing a pointer signifies that the pointer either points to a valid memory address or is initialized to a **"nil"** value. The **"nil"** value indicates that it presently doesn't reference any memory location.
- The assignment of a pointer means attribute it either the memory address of a particular variable or the address of a newly memory space after its creation.
- When a pointer is no longer used, it is recommended to reset it to **"nil"**.
- It is crucial to check the value of the pointer before using it.



1.3.4. Copying a pointer

Copying or duplicating a pointer to another is a common operation in programming, especially useful when working with dynamic data structures or functions. In algorithmics, this operation involves assigning a pointer the same memory address as another pointer, thus allowing multiple pointers to reference the same memory location.



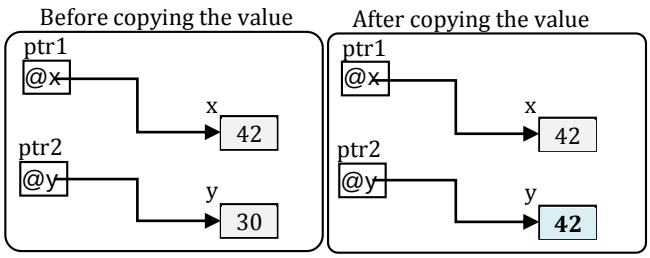
1.3.5. Copy of pointed value

The operation of "copy the value pointed by a pointer" consists in creating a copy of the data stored at a memory address referenced by a pointer to another memory location. In other words, this is duplicating the value pointed by a pointer in a distinct memory area. This process can be done by dereferencing the pointer using the indirection operator (*).

For example, if ptr1 points to an integer variable x, ptr2 is another pointer, and y is additional integer variable, the statement *ptr2 ← *ptr1; copies the value of x to the memory address pointed by ptr2. Similarly, the statement y ← *ptr1; copies the value of x into the variable y, ensuring that the memory space pointed by ptr2 and the variable y contain the same value as that pointed to by ptr1.

Syntax

In Algorithmics	In C language
ptr2* ← ptr1* ;	*ptr2 = *ptr1 ;



Note : The pointed values become the same but the pointers are different.

1.4. Example of using the pointers

In this example, we will explore pointer operations by applying them to a Student structure. This structure contains a set of fields such as name, first name, date of birth and average. In addition, we will use an auxiliary structure, Date, to store birth date information. This example will illustrate how pointer operations can be used to efficiently manipulate structures and access to their members dynamically. The comments included in the code will provide detailed explanations for each step, thus allowing a thorough understanding of the interactions between pointers and structures in a practical context.

In Algorithmics

```
Algorithm Example_pointer_operations;

   // definition of Date Structure
   structure Date
      day : integer ;
      month : integer ;
      year : integer ;
   endstructure;

   // definition of Student structure
   structure Student
      name : string ;
      f_name: string ;
      birth_date : Date ;
      average : real ;
   endstructure;

Var   st1 : Student ; // Declaration of a Student structure instance (st1)
      ptrst1,ptrst2,ptrst3,ptrst4:*Student;//Declaration of 4 pointers to the Student structure
```

```
Begin                                /* **********     Main Algorithm     ********** */

   // reading information for student 1
   write("Enter the name of student 1: ");
   read(st1.name);

   write("Enter the first name of student 1: ");
   read(st1.f_name);

   write("Enter the birth date of student 1 (day, month and year): ");
   read(st1.birth_date.day,st1.birth_date.month,st1.birth_date.year);

   write("Enter the average of student 1: ");
   read(st1.average);

   ptrst1 ← @st1; //Initialization of ptrst1 pointer with the address of st1 structure

   ptrst2 ← nil; //Initialisation of ptrst2 pointer with nil
   ptrst3 ← nil; //Initialisation of ptrst3 pointer with nil
   allocate(ptrst2);//Allocate memory dynamically for a Student structure and assign its address to the pointer ptrst2

   // Reading information for student 2 via the ptrst2 pointer
   write("Enter the name of student 2: ");
   read((*ptrst2).name);

   write("Enter the first name of student 2: ");
   read((*ptrst2).f_name);

   write("Enter the birth date of student 2 (day, month and year): ");
   read((*ptrst2).birth_date.day,(*ptrst2).birth_date.month,(*ptrst2).birth_date.year);

   write("Enter the average of student 2: ");
   read((*ptrst2).average);

   ptrst3 ← ptrst1; // Assigning the ptrst1 pointer to the ptrst3 pointer

   allocate(ptrst4);//Allocate memory dynamically for a Student structure and assign its address to the pointer ptrst4
   *ptrst4←*ptrst2;//Assigning the value of the structure pointed by ptrst2 to the structure pointed by ptrst4

   // Displaying information of students via different pointers
   write("ptrst1 pointer points to student with information:");
   write("Name: ", ptrst1->name, "First name: ",ptrst1->f_name);
   write("Birth date: ",ptrst1->birth_date.day,"/",ptrst1->birth_date.month,"/",
           ptrst1->birth_date.year);
   write("Average: ", ptrst1->average);

   write("ptrst2 pointer points to student with information:");
   write("Name: ", ptrst2->name, "First name: ",ptrst2->f_name);
   write("Birth date: ",ptrst2->birth_date.day,"/",ptrst2->birth_date.month,"/",
           ptrst2->birth_date.year);
   write("Average: ", ptrst2->average);

   write("ptrst3 pointer points to student with information:");
   write("Name: ", ptrst3->name, "First name: ",ptrst3->f_name);
   write("Birth date: ",ptrst3->birth_date.day,"/",ptrst3->birth_date.month,"/",
           ptrst3->birth_date.year);
   write("Average: ", ptrst3->average);

   write("ptrst4 pointer points to student with information:");
   write("Name: ", ptrst4->name, "First name: ",ptrst4->f_name);
   write("Birth date: ",ptrst4->birth_date.day,"/",ptrst4->birth_date.month,"/",
           ptrst4->birth_date.year);
   write("Average: ", ptrst4->average);
```

```
   // Freeing dynamically allocated memory
   //free(ptrst1);// cannot be used because the pointer ptrst1 refers to a variable of type Student structure
                  // that has not been dynamically allocated.
   free(ptrst2); // freeing the allocated memory dynamically pointed by the ptrst2 pointer
   ptrst2 ← nil;
   // free(ptrst3); // Cannot be used because the pointer ptrst3 refers the same structure as ptrst1
   free(ptrst4); // freeing the allocated memory dynamically pointed by the ptrst4 pointer
   ptrst4 ← nil;
End.                                /* **********     Main Algorithm     ********** */
```

In C language

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

   // definition of Date Structure
   struct Date {
      int day;
      int month;
      int year;
   };

   // definition of Student structure
   struct Student {
      char name[20];
      char f_name[20];
      struct Date birth_date;
      float average;
   };

int main() {                            /* **********     Main Program     ********** */
   // 1. Dynamic memory allocation and deallocation
   // Declaration of a Student structure instance (st1) without dynamic allocation
   struct Student st1;

   // reading information for student 1
   printf("Name of student 1: ");
   scanf("%s", st1.name);

   printf("First name of student 1: ");

   scanf("%s", st1.f_name);

   printf("Birth date of student 1 (day month year): ");
   scanf("%d%d%d",&st1.birth_date.day,&st1.birth_date.month,&st1.birth_date.year);

   printf("Average of student 1: ");
   scanf("%f", &st1.average);

   // 2. Access to the pointed value
   // Declaration of a pointer to Student structure ptrst1 and initialization with
   // the address of st1
   struct Student* ptrst1 = &st1;

   // 3. Initializing and assignment of a pointer
   // Declaration of other pointers (ptrst2,ptrst3) and dynamic allocation (ptrst2)
   struct Student* ptrst2 = NULL;
   struct Student* ptrst3 = NULL;
   ptrst2 = malloc(sizeof(struct Student));

   // Inputting information for student 2 via the ptrst2 pointer
   printf("Name of student 2: ");
   scanf("%s", (*ptrst2).name);

   printf("First name of student 2: ");
   scanf("%s", (*ptrst2).f_name);

   printf("Birth date of student 2 (day month year): ");
```

```
    scanf("%d %d %d", &(*ptrst2).birth_date.day, &(*ptrst2).birth_date.month,
           &(*ptrst2).birth_date.year);

    printf("Average of student 2: ");
    scanf("%f", &(*ptrst2).average);

    // 4. Copying a pointer
    // Assigning the address of the ptrst1 pointer to the ptrst3 pointer
    ptrst3 = ptrst1;

    // 5. Copy of pointed value
    // Dynamic allocation and copying of the structure pointed by ptrst2 to a new
    // structure ptrst4
    struct Student* ptrst4 = malloc(sizeof(struct Student));
    *ptrst4 = *ptrst2;

    // Displaying information of students via different pointers
    printf("ptrst1 pointer points to student with information:\n");
    printf("Name: %s\tFirst name: %s\t", ptrst1->name, ptrst1->f_name);
    printf("Birth date: %d/%d/%d\n",ptrst1->birth_date.day,ptrst1->birth_date.month,
            ptrst1->birth_date.year);
    printf("Average: %.2f\n\n", ptrst1->average);

    printf("ptrst2 pointer points to student with information:\n");
    printf("Name: %s\tFirst name: %s\t", ptrst2->name, ptrst2->f_name);
    printf("Birth date: %d/%d/%d\n",ptrst1->birth_date.day,ptrst2->birth_date.month,
            ptrst2->birth_date.year);
    printf("Average: %.2f\n\n", ptrst2->average);

    printf("ptrst3 pointer points to student with information:\n");
    printf("Name: %s\tFirst name: %s\t", ptrst3->name, ptrst3->f_name);
    printf("Birth date: %d/%d/%d\n",ptrst3->birth_date.day,ptrst3->birth_date.month,
            ptrst3->birth_date.year);
    printf("Average: %.2f\n\n", ptrst3->average);

    printf("ptrst4 pointer points to student with information:\n");
    printf("Name: %s\tFirst name: %s\t", ptrst4->name, ptrst4->f_name);
    printf("Birth date: %d/%d/%d\n",ptrst4->birth_date.day,ptrst4->birth_date.month,
            ptrst4->birth_date.year);
    printf("Average: %.2f\n\n", ptrst4->average);

    // Freeing dynamically allocated memory
    //free(ptrst1);//cannot be used because the pointer ptrst1 refers to a variable
                   //of type Student structure that has not been dynamically allocated.

    free(ptrst2);
    ptrst2 = NULL;
    // free(ptrst3); // Cannot be used because the pointer ptrst3 refers the same
                     //structure as ptrst1
    free(ptrst4);
    ptrst4 = NULL;

    return 0;
}                              /* **********     Main Program     ********** */
```

2. Linked lists

2.1. Introduction

Linked lists are fundamental data structures in algorithmics and programming, used to store and organize data dynamically. Unlike static arrays, linked lists provide enhanced flexibility in accommodating insertions, deletions, and modifications of elements. They are constructed from a sequence of elements called nodes, interconnected by links (or pointers). This dynamic Node allows linked lists to easily adapt to changes in data size and structure, making them a preferred choice in many computer applications.

Linked lists were born out of the need to overcome the limitations of static arrays, particularly the difficulty of managing data whose size is unknown in advance or may fluctuate during program execution. Thanks to their dynamic structure, linked lists provide an elegant solution to these problems by allowing memory allocation as needed, avoiding space wastage and optimizing resource utilization.

However, their main drawback resides in their use of additional memory to store pointers, which can lead to inefficient memory usage in some cases. Additionally, sequential access to elements can result in poorer performance compared to static arrays in certain situations.

Despite these disadvantages, linked lists are widely used in many areas of computer science, including the development of text editors, web browsers, database management systems, and search and sort algorithms. Furthermore, linked lists allow the implementation of complex data structures such as stacks and queues, making them a versatile tool in software development.

In this chapter, we will explore in detail the fundamental concepts of linked lists in algorithmics, as well as their practical implementation in the C language. We will explore different methods to create, manipulate, and traverse linked lists, focusing on basic operations such as insertion, deletion, and element search.

2.2. Definition of linked lists

Definition 1

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.

Definition 2

A linked list is a finite set of elements (nodes) of evolving linear structure; of arbitrary size.
The list is defined at least by a link (pointer) to its first node.
Each node in the list is defined by : His value, link (pointer) to the next node (except the last one).

Definition 3

A linked list is a data structure that groups a set of data of the same type and arbitrary size. Access to the elements of the list is done sequentially. From each element, one can access the next element.

2.3. Graphical representation of a linked list

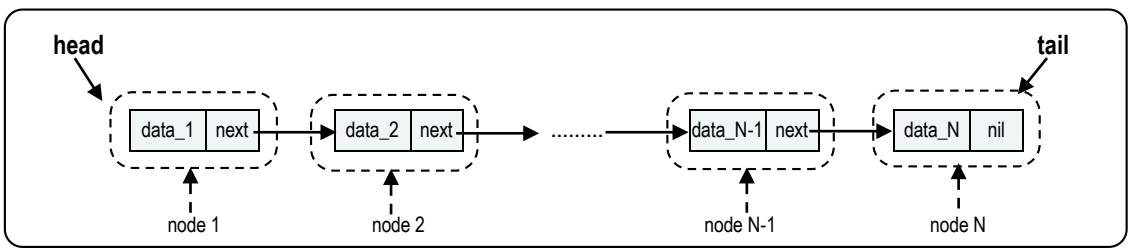


Figure 1 : Singly linked list

Where :

- **Node :** A node in a linked list is a structure that contains two components:
  - **Data:** It holds the actual value or data associated with the node.
  - **Next :** It stores the link (pointer) to the next node in the sequence.
- **Head and Tail:** The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to "**nil**", indicating the end of the list. This node is known as the tail node.

2.4. Types of linked lists

There are three main types of linked lists : Singly, doubly or circular linked lists.

2.4.1. Singly linked lists

Singly linked lists, also known as simply linked lists, are a type of linked list where each node contains a data element and a link (or pointer) to the next node in the sequence. The last node in the list points to **nil**, indicating the end of the list. Traversal a singly linked list is possible only in one direction, typically from the head (the first node) to the tail (the last node). The singly linked list is shown in Figure 1.

2.4.2. Doubly linked lists

Doubly linked lists are similar to singly linked lists, but each node in a doubly linked list contains links (pointers) to both the next and previous nodes in the sequence. This bidirectional linkage allows for traversal in both forward and backward directions, providing more flexibility in accessing and manipulating elements within the list. However, doubly linked lists require more memory due to the additional pointers in each node. The doubly linked list is shown in Figure 2.

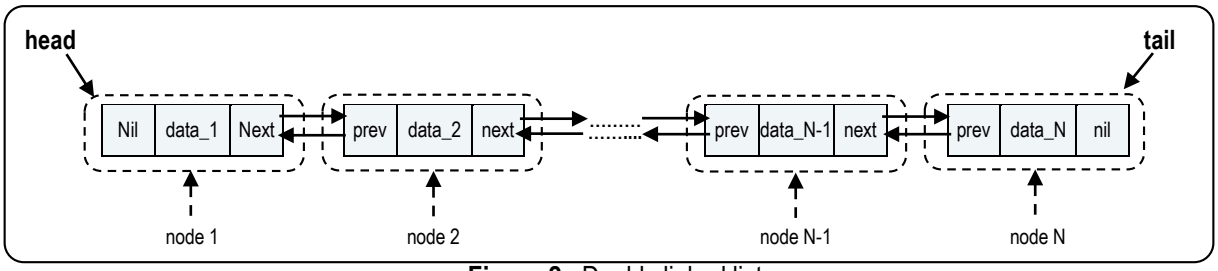


Figure 2 : Doubly linked list

2.4.3. Circular linked lists

Circular linked lists are variants of singly (or doubly) linked lists. In the case of singly linked lists the next of the last node becomes the first node witch forming a circular structure. In addition, in the case of doubly linked lists, the previous of first node becomes the last node. This circular linkage eliminates the concept of an "end" of the list, allowing for continuous traversal without reaching a **nil** pointer. Circular linked lists find applications in situations where data needs to be accessed cyclically, such as round-robin scheduling algorithms or representing cyclic data structures like rings or loops. The circular singly linked list is shown in Figure 3.

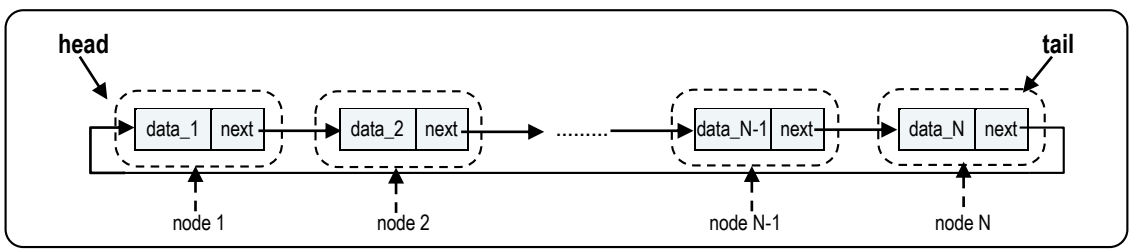


Figure 3 : Circular singly linked list

Note : In this course, we only study the **singly linked lists.**

2.5. Basic operations on Linked Lists

The basic operations on the liked lists are :

2.5.1. Insertion: this operation enables the addition of new data to the linked list, incorporating elements into the current data structure, whether at the beginning, middle, or end of the list.

2.5.2. Deletion: this operation enables to remove of data from the list. It provides the capability to delete elements from the existing data structure, whether at the beginning, middle, or end of the list.

2.5.3. Searching: this operation enables searching for a specific value in a linked list. It involves traversing the list from the first (head) node until the value is found or the end of the list is reached.

2.6. Representation of linked lists

A linked list can be represented in two forms: static or dynamic. The difference between these two representations is the way in which memory is allocated to store the elements (nodes) of the list.

- **Static representation :** in this representation a static linked list uses an array to store its nodes. This means that the size of the list must be determined in advance and cannot be changed dynamically.
- **The dynamic representation** : this representation involves the allocation of memory for each node dynamically, which consists of a value field and a pointer field to the next node ; The size of the list can be changed during the program's execution. The dynamic linked list is often more flexible and easier to use than the static linked list.

Notes :
- In this course, we only study the **dynamic representation of singly linked lists.**
- In C language, the nodes can be dynamically allocated using the **malloc**() function.

2.6.1. Dynamic representation of singly linked lists

A dynamic linked list is a list of elements (nodes) whose number evolves dynamically during the algorithm's execution.

- The nodes are of the same type.
- Each node is a structure with two fields : value or data field and next node (pointer) field.
- Each node (except the last one) is linked to the next one by a pointer.
- The beginning of the list is marked by a pointer to the first node.
- The end of the list is marked by the "nil" value of the field “next” of the last node.

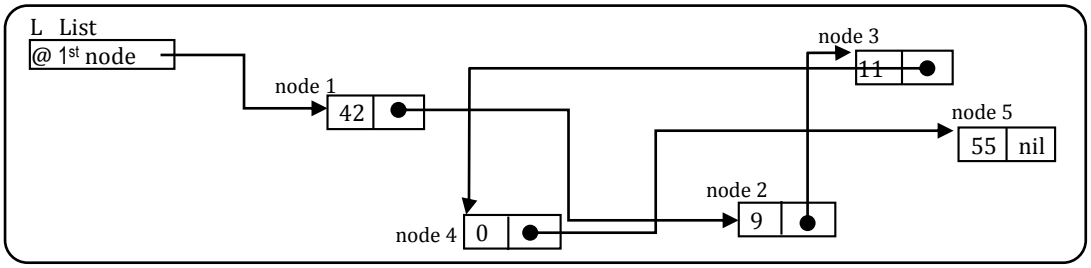


Figure 4 : Memory state of the dynamic singly linked list of integers 42, 9, 11, 0, and 55

2.7. Definition (declaration) of the dynamic linked list

There are several declarations to represent dynamic linked lists, among them we can mention :

Definition 1 :

A linked list is defined, only, by :
- A pointer to its first node (head).

Definition 2 :

A linked list is defined by :
- A pointer to its first node (head), and
- A pointer to its last node (tail).

Definition 3 :

A linked list is defined by:
- A pointer to its first node (head).
- A pointer to its last node (tail).
- Its size (the number of its elements).

Note : in this course, we use the **first** definition.

Definition 1

In Algorithmics

```
Type Pointer = * Node ;
structure Node
  value : value_type ;
  next : Pointer ;
endstructure ;
Type List = *Node ;

Var L : List ;
```

In C Language

```
struct Node {
   value_type value ;
   struct Node* next ;
} ;
typedef struct Node Node ;
typedef Node* List ;

List L ;
```

Note :

The general syntax to define a new simple type is as follows:
- In algorithmic : Type Name_NewType = Existing_Type ;
- In C language : typedef Existing_Type Name_NewType ;

Where :
- Existing_Type : represents the existing data type used to define a new type.
- Name_NewType : is the name of the new defined type.

For example :

To define a new type (called Pointer) that points to Node structure we use :
- In algorithmics : Type Pointer = *Node ;
- In C langage : typedef Node* Pointer ;

2.8. Operations on dynamic singly linked lists (case of the list of integers)

2.8.1. Initialization of the dynamic singly linked list

This operation initializes a dynamic linked list.

In Algorithmics

```
Procedure Initialize_list (var L : List)
Begin
   L ← nil ;
End ;
```

In C Language

```
void Initialize_list (List *L){
   (*L) = NULL ;
}
```

The procedure takes as input a variable « L » of type **List** as a parameter passed by address. It then sets the variable L to **nil**, which means it does not point to any element of the linked list. In other words, this procedure creates an empty dynamic linked list, ready to receive elements later on. It is often used at the beginning of a program or a function that uses a dynamic linked list, to ensure that the list is initialized correctly before any use.

2.8.2. Empty or non-empty test of a dynamic singly linked list

This operation allows to check if the linked list is empty or not.

In Algorithmics

```
Function Is_empty_list(L : List) of Boolean
Begin
   return (L = nil) ;
End ;
```

In C Language

```
int Is_empty_list(List L){
   return (L == NULL) ;
}
```

This function "Is_empty_list" takes as input a dynamic linked list and returns a **boolean** indicating whether the list is empty or not. It returns "true" if the list «L» is empty, otherwise it returns "false".

2.8.3. Determination of the length of a dynamic singly linked list

This function calculates the length of a dynamic linked list by traversing all the nodes of the list and counting their number.

In Algorithmics

```
Function Length_list (L : List) of integer
Var number : integer ;
    current : pointer ;
Begin
   current ← L ;
   number ← 0 ;
   while (current ≠ nil) do
      current ← current*.next ;
      number  ← number + 1 ;
   endwhile

   return (number) ;
End ;
```

In C Language

```
int Length_list(List L){
   List current = L ;
   int number = 0 ;
   while (current != NULL) {
      current = current->next ;
      number ++ ;
   }
   return (number) ;
}
```

The function takes as input a single formal parameter which is a pointer to the first node (head) of the linked list and returns an integer corresponding to the number of nodes present in the list.

Here is how this function works:

- We initialize a pointer current with the value of the head of the list « L ».
- We initialize a counter number to 0.
- As long as the current pointer is not **NULL** (meaning it points to a valid node of the list), we move forward in the list by moving to the next node and incrementing the counter number.
- When the current pointer becomes **NULL**, it means we have reached the end of the list. We then return the value of the counter "**number**" which corresponds to the number of nodes traversed.

2.8.4. Insertion at the beginning of a dynamic singly linked list

This function is used to insert an element at the beginning of a dynamic singly linked list.

In Algorithmics

```
Procedure Insert_beginning_list (var L : List, x : integer)
Var current : pointer ;
Begin
   Allocate (current) ;
   current*.value ← x ;
   current*.next  ← L ;
   L ← current ;
End ;
```

In C Language

```
void Insert_beginning_list(List *L, int x){
Node* current = malloc(sizeof(Node)) ;
   (*current).value = x ; //or current->value = x ;
   current->next = (*L) ;
   (*L) = current ;
}
```

It takes as a formal parameter (passed by address) a linked list "L" and an integer "x" and inserts a new element containing the value "x" at the beginning of the linked list "L".

The function starts by allocating memory for a new node, whose address is stored in the "current" variable of pointer type. Then, the value of the element to be inserted is stored in the "value" field of the "current" pointer. The "next" field of the "current" pointer is initialized with the existing list "L". Finally, the pointer "L" is updated to point to the new node, so that it becomes the first element of the list. In summary, the function creates a new node, inserts it at the beginning of the list, and modifies the head of the list to point to this new node.

2.8.5. Insertion at the end of the dynamic singly linked list

This function inserts a new node at the end of a dynamic singly linked list.

In Algorithmics

```
Procedure Insert_end_list(var L: List, x: integer
Var current, p: pointer;
Begin
   Allocate (current);
   current*.value ← x ;
   current*.next ← nil;
   if (Is_empty_list(L)) then
      L ← current ;
   else
      p ← L;
      while (p*.next ≠ nil) do
         p ← p*.next;
      endwhile
      p*.next ← current;
   endif
End ;
```

In C Language

```
void Insert_end_list(List* L, int x){
   Node *current = malloc (sizeof (Node));
   current->value = x;
   current->next = NULL;

   if (Is_empty_list(*L))
      (*L) = current;
   else {
      Node* end = (*L);
      while ((end->next) != NULL) {
         end = end->next;
      }
      end->next = current;
   }
}
```

Here is how the function works in detail:

- It takes two parameters as input: the linked list "L" (passed by address) and the value x (an integer) to add to the end of the list.
- It allocates memory for a new node using the **Allocate** function and stores its address in the current pointer.
- It assigns the value x to the "value" field of the space pointed by current.
- It assigns **nil** to the next field of the space pointed by current ; the new node will be the last element of the list.
- If the list L is empty, the function assigns the pointer current to L, which means that the new node is the single element in the list.
- Otherwise, the function traverses the list until it reaches the last node using a While loop and a variable p, which is initialized to L.
- Once it reaches the last node of the list (the node whose next pointer is **nil**), it assigns the pointer current to "p*.next", which means that the new node is added to the end of the list.

2.8.6. Insertion in the middle at position "k" of a dynamic singly linked list

This function aims to insert an integer value "x" at a given position "k" in a dynamic linked list "L"

In Algorithmics

```
Procedure Insert_middle_list(var L: List, x: integer, k: integer)
/* we assume that the insertion position "k" is between 1 and the size of the list (with size>=2)*/
Var previous, current,p : pointer;
     pos : integer;
Begin
   previous ← L;
   pos ← 1;
   current ← previous*.next;

   while(pos≠k-1)do
      previous ← current ;
      current ← current*.next ;
      pos ← pos + 1 ;
   endwhile

   Allocate(p) ;
   p*.value ← x ;
   p*.next ← current ;
   previous*.next ← p ;
End ;
```

In C Language

```
void Insert_middle_list (List* L, int x, int k){
/* we assume that the insertion position "k" is between 1 and the size of the list (with size>=2)*/
   int pos=1;
   Node* previous=(*L);
   Node *current=previous->next;
   while (pos!=k-1){
      pos++;
      previous=current;
      current=current->next;
   }
   Node *p=malloc(sizeof(Node));
   p->value=x;
   p->next=current;
   previous->next=p;
}
```

The Insert_middle_list procedure allows inserting a new node into a dynamic linked list at a specific position k. To do this, we assume that the value of k passed as a parameter is between 1 and the size of the list, and the list size is greater than or equal to 2.

The insertion process begins by searching for the insertion position by traversing the linked list from its head until the desired position is reached. We use the following variables to perform this search:

- The variable "pos" is initialized to 1 to follow the current position when traversing the list.
- The pointer "previous" is initialized with the address of the list's head and points to the node that precedes the insertion position.
- The current pointer is used to traverse the list from the second node and points to the node following the insertion position.

Utilizing a while loop, the procedure iterates through the list until the current position (pos) equals k-1, which is the position just before we insert the new element. At each iteration, the "pos" variable is incremented, and the "previous" and "current" pointers are updated to point to the next nodes in the list.

Once the insertion position is reached, a new node "p" is dynamically allocated in memory using the "allocate" function (malloc in C language). The "x" value is assigned to the value field of the newly created "p" node. Then, the next field of the "p" node is initialized with the "current" pointer, which points to the current node at the insertion position. Finally, the next field of the "previous" node is updated to point to the newly created "p" node, thus inserting the new element between the "previous" and "current" nodes.

2.8.7. Deletion at the beginning of the dynamic singly linked list

This procedure takes as input a dynamic linked list L and deletes its first Node (at the beginning).

In Algorithmics

```
Procedure Delete_beginning_list (var L : List)
Var current : pointer ;
Begin
   if (Is_empty_list(L)) then
      write ("Deletion error : the list is empty ") ;
   else
      current ← L ;
      L ← L*.next ;
      free(current) ;
   endif
End ;
```

In C Language

```
void Delete_beginning_list (List *L){
   if (Is_empty_list(*L))
      printf("Deletion error : the list is empty \n") ;
   else{
      Node* current = (*L) ;
      (*L) = (*L)->next ;
      free(current) ;
   }
}
```

The function starts by calling the function "Is_empty_list" to check if the list is empty or not; if the list is empty, it displays an error message, otherwise, it assigns to the current pointer (Node to be deleted) the address of the first element of the list L. It updates the list L by pointing to the next Node in the list. Finally, it frees the memory allocated to the first element of the list (the space pointed to by current).

2.8.8. Deletion at the end of the dynamic singly linked list

This function allows to delete the last element of a dynamic singly linked list.

In Algorithmics

```
Procedure Delete_end_list (var L : List)
Var end,prev_end : pointer ;
Begin
   if (Is_empty_list(L)) then
      write ("Deletion error: the list is empty ") ;
   else
      if (L*.next=nil) then
         end ← L ;
         Initialize_list (L) ; // or : L ← nil ;
      else
         prev_end ← L ;
         end ← prev_end*.next;
         while (end*.next ≠ nil) do
            prec_fin ← fin;
            end ← end->next ;
         endwhile
         next_end*.next ← nil ;
      endif
      free(end);
   endif
End ;
```

In C Language

```
void Delete_end_list (List* L){
   if (Is_empty_list(*L))
      printf("Deletion error: the list is empty \n") ;
   else{
      Node *end, *prev_end;
      if ((*L)->next==NULL) {
         end=(*L);
         Initialize_list(L);
      }
      else {
         prev_end = (*L);
         end = prev_end->next;
         while (end->next!=NULL) {
            prev_end = end;
            end = end->next ;
         }
         prev_end->next=NULL;
```

```
        }
        free(end);
    }
}
```

The procedure is define with a single formal parameter "L" of type List (passed by address), representing a pointer to the first element of the list.

Initially, we verify whether the list is empty. If it is, we display an error message and exit the procedure. In case the list isn't empty, we check if it contains only one element. If so, we assign its address to the "end" pointer and reset the list (list becomes empty).

If the list contains more than one element, we initialize the "prev_end" and "end" pointers by pointing them to the first and second element of the list, respectively. We then traverse the list by advancing the pointers until the "end" pointer points to the last element of the list. After identifying the last node, the "prev_end" pointer points to the second-to-last node of the list, which becomes the last node; subsequently, we update its next field to nil. Finally, we free the memory space occupied by the "end" pointer using the "free" function.

2.8.9. Deletion in the middle of the dynamic singly linked list

This function allows to delete the k^{th} element of the dynamic singly linked list.

In Algorithmics

```
Procedure Delete_middle_list (var L: List, k: integer)
/* we assume that the deletion position k is between 2 and the size-1 of the list (with size>=3)*/
Var prev_del,del: pointer ;
    index_prev : integer;

Begin
  prev_del ← L;
  index_prev ← 1 ;
  del ← prev_del*.next ;
  while (index_prev≠k-1) do
    prev_del ← del ;
    del ← del*.next;
    index_prev ← index_prev + 1;
  endwhile

  prev_del*.next ← del*.next;
  free(del);
End ;
```

In C Language

```
void Delete_middle_list (List *L, int k) {
  Node *del,*prev_del;
  int index_prev;

  prev_del = (*L);
  index_prev = 1;
  del = prev_del->next;
  while (index_prev != k-1){
    prev_del = del;
    del = del->next;
    index_prev++;
  }
  prev_del->next = del->next;
  free(del);
}
```

This function takes as formal parameters a dynamic linked list L (passed by address) and an integer "k" (passed by value) which represents the index of the element to be deleted.

With while loop, we traverse the list to find the element to be deleted using two pointers: "del" points to the element to be deleted and "prev_del" points to the element before the one to be deleted, along with a counter "index_prev" to track the index of the element before the one to be deleted.

Once the traverse is complete, we update the list (the next of the element pointed by "prec_del" pointer becomes the next of the element pointed by "del" pointer) and free the space allocated by the deleted element (pointed by the "del" pointer).

2.8.10. Filling (reading) a dynamic singly linked list

This procedure allows the filling a dynamic singly linked list with "N" values.

In Algorithmics

```
Procedure Read_list (var L: List, N: integer)
Var i, val, choice, pos: integer;
Begin
   for (i←1 to N) do
      write ("Enter the ", i,"th value");
      read(val);
      write ("Choose the operation to apply: ");
      write ("1: to insert at the beginning ");
      write ("2: to insert in the middle ");
      write ("3: to insert at the end ");

      write ("Enter your choice : ");
      repeat
         read(choice);
         if(((choice≠1)and(choice≠2)and(choice≠3))or((choice=2)and(i<=2)))then
            write("Re-enter your choice : ");
         endif
      Until(((choice=1)or(choice=2)or(choice=3))and((choice!=2)or(i>2)));

      According to (choice) do
         1: Insert_beginning_list(L, val);
         2: write ("Enter the insertion position: ");
            repeat
               read(pos);
               if((pos<=1)||(pos>=i))then
                  printf("Re-enter the insertion position: ");
               endif
            Until((pos>1)and(pos<i));
            Insert_middle_list(L, val, pos);
         3: Insert_end_list(L, val);
      endaccording;
   endfor
End ;
```

In C Language

```
void Read_list (List* L, int N){
   int i, val, choice, pos;

   for (i=1; i<=N; i++) {
      printf("Enter the %d th value ", i);
      scanf("%d", &val);
      printf("Choose the operation to apply : \n");
      printf("1: to insert at the beginning \n");
      printf("2: to insert in the middle \n");
      printf("3: to insert at the end \n");

      printf("Enter your choice : ");
      do{
         scanf("%d",&choice);
```

```
        if(((choice!=1)&&(choice!=2)&&(choice!=3))||((choice==2)&&(i<=2)))
          printf("Re-enter your choice : ");
      }while(((choice!=1)&&(choice!=2)&&(choice!=3))||((choice==2)&&(i<=2)));

      switch (choice){
        case 1: Insert_beginning_list(L,val); break;
        case 2: printf("Enter the insertion position: ");
                do{
                  scanf("%d",&pos);
                  if((pos<=1)||(pos>=i))
                    printf("Re-enter the insertion position: ");
                }while((pos<=1)||(pos>=i));
                Insert_middle_list(L, val, pos);
                break;
        case 3: Insert_end_list(L,val); break;
      }
    }
}
```

This function takes two formal parameters: the linked list "L" (passed by address) and the number of values to insert in the linked list.

For each element to insert, the function asks the user to enter the value, then offers three choices of insertion operations: insertion at the beginning, in the middle, or at the end of the list.

The user must choose one of these options by entering the corresponding number. If the user enters an invalid choice, the function asks them to enter a valid choice again.

The function uses a switch-case to call the corresponding insertion function based on the user's choice. If the user has chosen insertion in the middle, the function also asks for the position to insert the element.

2.8.11. Print (Displaying) a dynamic singly linked list :

This operation allows the display of elements in dynamic singly linked list.

In Algorithmics

```
Procedure Print_list (L: List)
Var  currant : Pointer;
Begin

   currant ← L;
   write ("The values of the list are : ");

   while (currant ≠ nil) do
      write (currant*.value) ;
      currant ← currant*.next;
   endwhile
End ;
```

In C Language

```
void Print_list (List L){
   Node* currant=L;

   printf ("The values of the list are : ");

   while (currant != NULL) {
      printf("%d",currant->value);
      currant =currant->next;
   }
}
```

The procedure is defined with a single formal parameter: the linked list L, passed by reference. Utilizing a while loop and initializing the current pointer with the address of the first node in the list, we iterate through the list until its end is reached. During each iteration of the loop, we print the value of the current node and then update the "current" pointer to refer to the next node in the list.

2.8.12. Search in a dynamic linked list:

This function checks the existence of a given value "x" in the linked list.

In Algorithmics

```
Function Search (L: List, x: integer) of Pointer
Var current : Pointer ;
Begin
   current ← L ;
   while ((current ≠ nil)and(current*.value ≠ x)) do
      current ← current*.next ;
   endwhile
   return (current);
End ;
```

In C Language

```
Node* Search (List L, int x){
   Node * current = L;
   while((current!=NULL)&&(current->value != x)){
      current = current->next;
   }
   return(current);
}
```

This function takes two parameters the linked list "L" and the value searched for "x", and it returns a pointer to the element of the linked list containing the first occurrence of the searched value "x" if it exists, otherwise the function returns "nil".

2.9. Example

Let the following algorithm :

In Algorithmics

```
Algorithm Test_list;
   Type Pointer = *Node ;
   structure Node
      value : integer ;
      next : Pointer ;
   endstructure ;

   List = *Node ;

Var number : integer ;
   L: List;
   /* Here we add all the previous procedures and functions. */
Begin
   Initialize_list(L);
   Write ("Enter the number of elements to insert in the list ");
   Read(number);
   Read_list(L,number);
   Print_list(L);
   Delete_end_list(L);
   Delete_beginning_list(L);
   Delete_middle_list(L,3);
   Print_list(L);
End.
```

In C Language

```c
#include <stdio.h>
#include <stdlib.h>
   struct Node {
      int value ;
      struct Node* next ;
   } ;
   typedef struct Node Node ;
   typedef Node* List ;
   void Print_list (List L);
   void Read_list (List* L, int N);
   void Delete_middle_list (List *L, int k);
   void Delete_end_list (List* L);
   void Delete_beginning_list (List *L);
   void Insert_middle_list (List* L, int x, int k);
   void Initialize_list (List *L);
   int Is_empty_list(List L);
   int Length_list(List L);
   void Insert_beginning_list(List *L, int x);
   void Insert_end_list(List* L, int x);
int main(){
   int n;
   List L ;
   printf("Enter the number of values to be added into list : ");
   scanf("%d",&number);
   Read_list(&L,number);
   Print_list(L);
   Delete_end_list (&L);
   Delete_beginning_list (&L);
   Delete_middle_list (&L,3);
   Print_list(L);
   return 0;
}
   /* Here we add all the previous procedures and functions. */
```

This C program implements various manipulation operations on dynamic singly linked lists. It begins by defining a structure called "Node" to represent each element of the list. Each node contains two fields: "value" (of integer type) to store the value of the element and "next" (a pointer) to point to the next element in the list.

Subsequently, functions and procedures are defined to perform different operations such as displaying the list, reading values from input, deleting an element from the end, beginning, or middle of the list, inserting an element at the beginning, middle, or end of the list, initializing the list, checking if the list is empty, and calculating its length.

The program starts by initializing an empty list, then prompts the user to specify the number of elements they want to add to the list. Next, the program uses the "Read_liste()" function to insert the specified number of elements into the list. For example, if the user wants to add 6 elements with the following values: 10, 18, 5, 2, 47, and 36, the program adds them in this order: 10 at the beginning, 18 at the end, 5 at the end, 2 at position 2, 47 at the beginning, and 36 at position 4.

After inserting the elements, the program uses the "Print_list()" procedure to display the list. Then, it performs three deletion operations: deletion from the end, deletion from the beginning, and deletion from the middle (position 3). Finally, the modified list is displayed again using the "Print_list()" procedure.

These different steps of this program are illustrated by the following figures:

Figure 1 : state of the list after its initialization

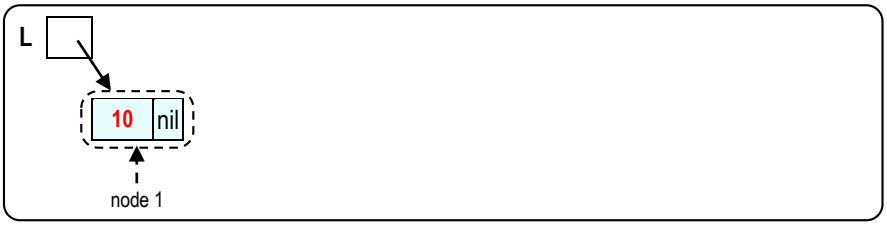


Figure 2 : state of the list : call of Read_lis() and after insertion 10 (at beginning)

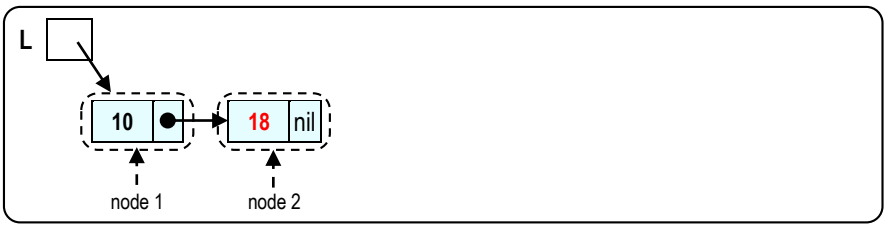


Figure 3 : state of the list : call of Read_lis() and after insertion 18 (at end)

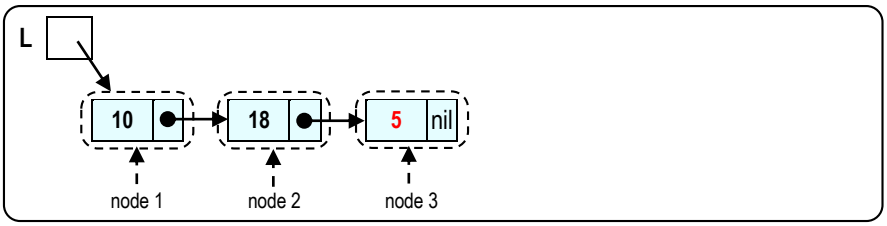


Figure 4 : state of the list : call of Read_lis() and after insertion 5 (at end)

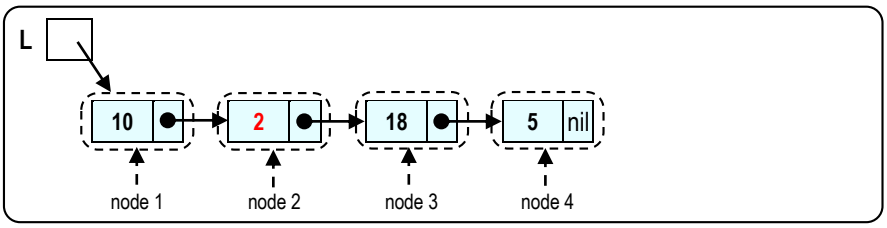


Figure 5 : state of the list : call of Read_lis() and after insertion 2 (at position 2)

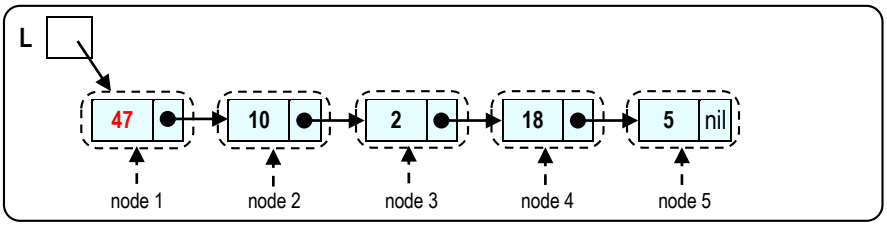


Figure 6 : state of the list : call of Read_lis() and after insertion 47 (at beginning)

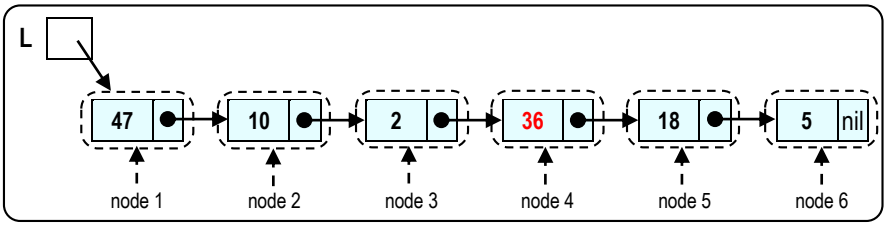


Figure 7 : state of the list : call of Read_lis() and after insertion 36 (at position 4)

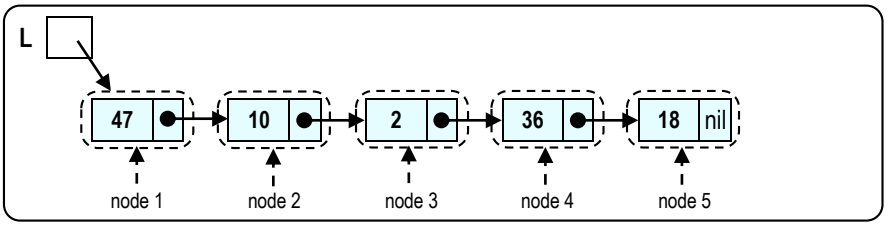


Figure 8 : state of the list : after call of Delete_end_list()

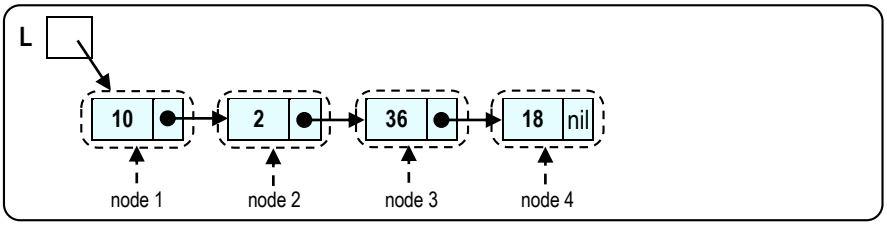


Figure 9 : state of the list : after call of Delete_beginning_list() procedure

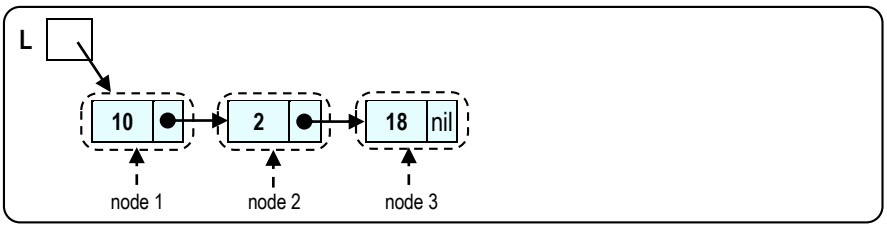


Figure 10 : state of the list : after call of Delete_middle_list() procedure (position 3)