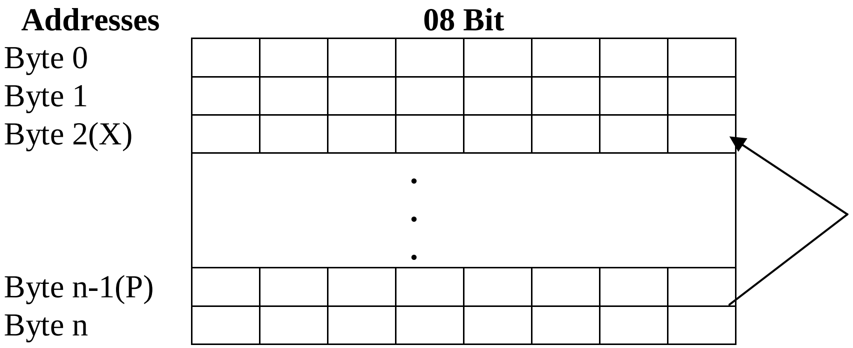
Chapter 3: Linked lists

1. Introduction

- Central memory is made up of a very large number of bytes.
- Each byte is identified by a number called the byte address.



- Each variable in memory occupies contiguous bytes, that is, bytes that follow one another.

Example :

float x;

In C++:
- A float occupies 4 consecutive bytes. The address of the variable is the address **of its first byte** .
- We can know the address of the variable x by the operator **&** .

float x;
p = &x ; //address of variable x: address of its first byte
The variable **p** contains a value which is **the address** of the variable **x**

2. Memory allocation

- The algorithms (programs) essentially consume two resources:
  - ✓ Execution time.
  - ✓ The reserved memory space.
- There are two types of allocation or reservation of memory space:
  - ✓ Static allocation.
  - ✓ Dynamic allocation.

2.1.Static allocation :

- The allocation of memory space is made **before execution** of the program (after compilation).
- This is therefore the case of simple and array type variables.

Example :
X: real;
A: integer;
T[50]: integer array;
Student_List[1500]: Student Table;
X, A, T, Student_List are ***static variables***

2.2.Dynamic allocation:

- Space is allocated as the program is executed.
- To be able to make this type of allocation, the user must have both operations: ***allocation*** and ***release*** of memory space.
- The majority of programming languages offer this possibility.

The operating system provides a part of the Central Memory for this purpose called the TAS (table allocation system).

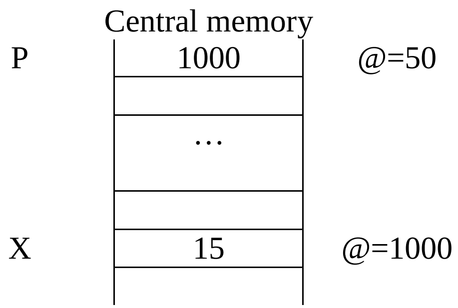
Example :
int X;
P=&X ;
X is a dynamic variable

3. The pointer type

3.1.Definition :

- In programming, a **pointer** is a variable containing a memory address.

Example :



x=15
P a pointer contains the address of x in memory
P= 1000

(*)Symbol for the pointer type

3.2.Declaration of a pointer:

In algorithmic	In C++ language
<Variable name>: * Variable type;	***Variable type * <Variable name>;***
Examples :	**Examples :**
p: ***** integer;	int * P ;
Q:*real;	float * Q ;

Example 1:

```
#include <iostream>
int main ( )
{
int x=4; //declaration of an integer variable x
int *p ; //declaration of a pointer variable p
p=&x; //p points to x (p contains the address of x)
cout <<*p ; // display the contents of x
getchar();
return 0;
}
```

Result : 4

Example 2:

```
#include <iostream>
int main ( )
{
int x=4; //declaration of a variable x
int *p ; //declaration of a pointer p
P=&x ; //p points to x (p contains the address of x)
*p=15; // the value 4 of x is replaced by 15
cout << x ;                    // print the contents of x
Return 0;
}
```

Result: 15

Example 3:

```
#include <iostream>
int main ( )
{
int x=4; //declaration of a variable x
```

```
int *p ; //declaration of a pointer p
P=&x; //p points to x (p contains the address of x)
*p= -3; // the value 4 of x is replaced by -3
X = X*3;
cout<< x                    // print the contents of x
Return 0;
}
```

Result : -9

3.3.Pointer operation

a) Dynamic allocation:

Syntax :

In algorithmic	In C++
P= Allocate (type)	P = **new type** ;

- Allocation of a space of size specified by the type of P.
- The address of this space is rendered in the variable of pointer type P.

Examples:

- Allocation of a memory area for an integer:

```
int * p;
P = new int;
```

- Allocation of a memory area for an array of 10 integers:

```
int * tab;
tab = new int[10];
```

- To access the three boxes in our table we can do the following:

```
/* The first box */
(*tab)= 16;
/* The second box */
*(tab + 1) = 12;
/* The third box */
*(tab + 2) = 11;
```

b) Freeing a pointer:

Syntax :

In algorithmic	In C++
delete **(p)**	delete(P);

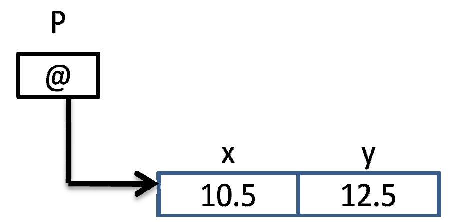
- **freeing** the memory space pointed by P

c) Pointer and record:

- To access a field of a record pointed to by a pointer P, we use the "->" notation:

Syntax : **P -> "field"**

Example :




```
Algorithm exp_PE
type
structure point
x: real ;
y: real;
end structure
P: * point;
Begin
P = Allocate (point) ; // memory space allocation
P -> x ←10.8;
P -> y ←12.5;
Write (p -> x); // displays 10.8
Delete (p); //freeing up memory space
END.
```

4. "List" data structure

4.1. Definition

- A list is a data structure consisting of a finite (possibly empty) sequence ***of elements of the same type*** .
- Each element in the list is ***identified*** according to their ***rank*** in the list.

<u>Example</u> :

a list of integers L = {11,-5,6,0}

the rank of element **6** is **3** .

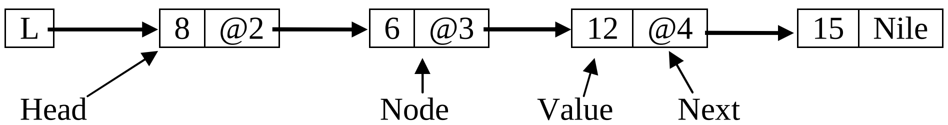
4.2.Representation of lists

- Two possible solutions:

1) **Contiguous representation:** we use contiguous cells when placing elements in a table.
2) **Linked (or chained) representation:** it consists of using pointers to connect the elements.

4.3.linked lists

- A **l**inear **l**inked **l**ist **LLL** is a set of links (dynamically allocated memory boxes) linked together.
- Schematically, it can be represented as follows:



- A **Node** is a structure with two fields:
  - ✓ **Value** field containing the information.
  - ✓ **Next** field giving the address of the next link.

- The first element address of an LLC is often called *head of the list.*
- Each Node is associated with an address. This address is stored in the next field of the previous Node.
- The next field of the last Node points to Nil (consists of the address which does not point to any Node).

4.4.Declaration:

- In algorithmic language:

type
Structure ***Name_Type_*** **Node**
Element : Typeqq;
Next : * ***Name_Type_*** **Node**;
End Structure ;
type
List: * ***Name_Type_*** **Node** ;// the list type which designates each Pointer // to a Node
L: List ; // the same meaning to L: * ***Name_Type_*** **Node**;
L, P, Q: List;

Typeqq: designates any type (int, float, person, student, Product, etc.).

The declarations:

L, P, Q: List; // means L, P, Q are pointers to Nodes

- In C++ language:

```
TypeDef
struct Name_Type_Link
{
Typeqq Element ;
Name_Type_Node * Next ;
};
typedef
Name_Type_ Node * List ;
List L, Q, head ; // equivalent to Name_Type_ Node * L, Q, head;
```

***Example* 1:** linked list of integers

type
Structure node
Ele: integer;
next: * node;
end structure

Type List: * node;

LE: List;

***Example* 2:** linked list of student

type
structure Student
…
end structure
type
Structure node
Ele: student;
next: * node;
end structure
Type List: * node ;

L:list;

***Example* 3:** linked list of people

Type

Structure Person

name: string

first name: string

age: integer

end structure

Type

Structure node

Ele: person;

next: * node;

end structure

Type List: * node;

***Noticed* :**

- Access to a field of a structure is done through the **point** for ordinary variables and through the " -> "for pointer type variables.

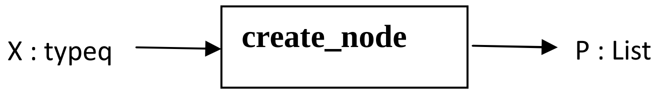
***Example* :**

X: Student; then X.name

Y: *Student; then Y->name.

4.5.List operations:

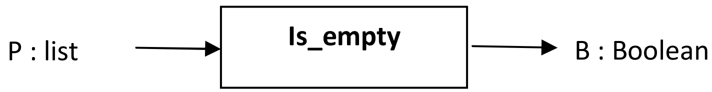
a) **create_node** : creates a new node containing the value x and returns a pointer containing its address.



Role : creates a new node

```
Function create_node (x: typeq): List
P: List // or P: * node
Begin
P ←Allocate (node); ………….(1)
P - > Ele ←x;…………………(2)
P -> next ←Nile; ………………(3)
Return (P);
END
```

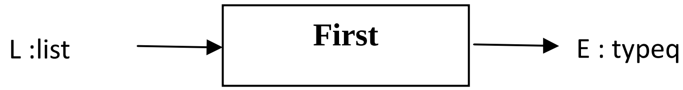
b) **Is_empty:** tests if the list is empty or not



Role : tests if the list is empty or not

```
Function Is_empty(L: List): Boolean
Begin
If (L=Nil) then
Return (true);
else
Return (false);
End if
END ;
```

c) **First** : returns the first element of the list L



Role : returns the first element of the list L

```
Function First  (L: list): typeq ;
Begin
If (L==Nil) then
Write ("the list is empty");
else
Return (L -> ele);
End if
END
```

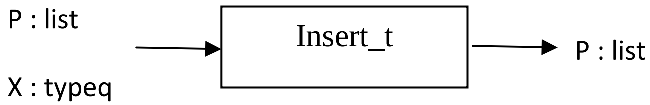
d) **Insertion of an element:** the insertion of a new element in the LLL list consists of:

1) First creation of the corresponding **node**,
2) Assignment of the value to the element fild,
3) Then the chaining of the new node with the list LLL is:
   - ✓ ***At the top of the list*** **:** To add the new node at the start of the list we must change the address of the head each time.
   - ✓ ***At the end of the list*** **:** To add the new node at the end of the list we must always keep two pointers: the head of the list and the tail (address of the last node) of the list.
   - ✓ ***In the middle of the list*** **:** To add the new node in the middle of the list we must first

   -Find its position (the address of the node which will precede it p and that of the node which will follow it s)

- Then cut the chaining between p and s.
- Now p points towards the new node and the new node points towards s.

<u>Inserting an element at the top of the list :</u>



Role : insert an element at the top of the list

Function Insert_t (x: typeq, L: List): List;
P: List; // or P: *node ;
Begin
P ←create_node (x);
P -> next ←L;
Return (P);
END ;

Procedure Insert_t (x: Element, **var** L: List);
P: List;
Begin
P ←create_node (x);
P -> next ←L;
L ← P;
END ;

e) **Rest (L: List):** returns the list L without the first element. It returns the address of the next element (node) in the list.

Role : returns the list L without the first element

Function rest (L: List): List
Begin
Return (L -> next);
END;

f) **List length:** returns the number of elements in list L.

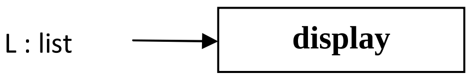
Role : returns the number of elements in list L

Function List_length (L: List): integer // recursive version.
Begin
If (L=Nil) **then**
Return (0);
else
Return (1 + **List_length** (Rest(L)));
End if
END

function List_length (L: List): integer **// iterative version.**
Current: List; // current: *node;
Nb: integer;
Begin
Number ←0; current ←L;
While current**!=** Nil **do**
Nb ←Nb+1;

```
current ←current ->next ;
end while
Return (Nb);
END;
```

g) display : Show elements in a list.



Role : display the elements of the list.

```
Procedure display (L: List) // recursive version.
Begin
If (L!=Nil) then
Write (first (L));
 else
Write (first(L));
Show(Rest (L));
End if
END ;
```

```
Procedure display (L: List);// iterative version.
Current: List; // current: *node
Begin
While (current != Nile) do
Write (current -> Ele);
current ←->next;
end while
END ;
```

h) **Deletion of an element:** Deletion consists first of breaking the chaining of the node concerned from the list, according to one of the three cases, then releasing this node:

- ✓ ***Remove first item from list:*** Changed the head of the list to point to the next node in the list.
- ✓ ***Delete an element in the middle of the list :*** Let
- ✓ ***Remove the last element from the list :*** the node before that in the queue will point to Nile.

Example:

```
procedure  remove_first (Var L: List)
P: List;
Begin
P ←L ;
L ←L->next ;
Release (P);
END;
```

2.6. Types of linked lists

a) Simple linked lists

b) Doubly linked lists

c) Simple circular linked lists