

Mail délégué :
imen.mano.03@gmail.com

Chapitre I : Rappels sur les Bases de l'Algorithmique

Chapitre II : Représentation des Données

II.1 Définition

Une structure de données (SDD) est une organisation des données pour en faciliter l'accès et la modification vis-à-vis d'un problème donné. Elle est définie par le type de base des composants ainsi que les opérations que l'on peut leur appliquer.

Quelques structures de données :

- Les structures séquentielles : les **tableaux**, **listes**, les **pires** et les **files**.
- Les structures arborescentes : les **arbres**.
- Les structures relationnelles : les **graphes**.
- Les structures à accès par clé : les **tables**.

II.2 Choix de la structure de données

La manière de représenter les données en mémoire doit respecter deux contraintes :

1. utiliser un minimum d'espace mémoire;
2. exécuter un nombre minimal d'instructions pour réaliser une opération.

Le choix d'un algorithme peut déterminer le type de la structure de données, et parfois l'adaptation d'une structure de données particulière peut déterminer quel algorithme utiliser.

Le choix de bonnes structures de données et de bons algorithmes peut faire la différence entre un programme s'exécutant en quelques secondes ou en quelques jours.

II.3 Représentation contiguë

Un **tableau** (*array* en anglais) est une structure de données qui consiste en un ensemble d'éléments ordonnés accessibles par leur indice (ou index). C'est une

structure de données de base que l'on retrouve dans chaque langage de programmation.

Tous les éléments d'un tableau doivent être du même type¹. L'accès à un élément par son indice est direct, cela s'explique par le fait que les éléments d'un tableau sont contigus dans l'espace mémoire. Ainsi, il est possible de calculer l'adresse mémoire de l'élément auquel on veut accéder, à partir de l'adresse de base du tableau et de l'indice de l'élément. L'accès est direct, comme il le serait pour une variable simple.

Exemple : Soit $E = \{3, 21, 6, 13, 68, 4, 29\}$ un ensemble d'éléments de type entier peut être représenté par un tableau T schématisé comme suit :

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
$T :$	3	21	6	13	68	4	29
	1	2	3	4	5	6	7

Tel que : 1, 2, ..., 7 représentent les indices et

a_1 représente l'adresse de $T[1]$

...

a_n représente l'adresse de $T[n]$

En général, a_i représente l'adresse de $T[i]$, en calculant l'adresse en mémoire du i ème élément on accède directement à l'élément $T[i]$ ainsi :

$$a_i = a_{i-1} + \text{<taille du type> ou encore } a_i = a_1 + (i-1) * \text{<taille du type>}$$

Remarque : <taille du type> en C est `sizeof(int)` si les éléments sont de type entier.

Les limites de la structure tableau sont :

- Vous devez connaître au moment de l'écriture de l'algorithme le nombre de données que devra recevoir le tableau, c'est-à-dire son nombre de cases.
- Un tableau étant représenté en mémoire sous la forme de cellules contiguës, les opérations d'insertion et de suppression d'éléments nécessitent de faire des décalages.
- L'insertion peut être impossible si le tableau est plein.

Cela fait donc beaucoup d'opérations et oblige certains langages fournissant de telles possibilités à implémenter leurs tableaux, non pas sous la forme traditionnelle (cellules adjacentes), mais en utilisant une représentation chaînée (liste chaînée).

¹ Dans certains langages (comme [Python](#), [APL](#), etc.), cette restriction n'existe plus.

Un cahier, dont le nombre de pages est fixé a priori, fournit une bonne image d'une variable statique.

II.4 Représentation chaînée

a- Les variables dynamiques :

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible.

Il nous faut donc un moyen de gérer manuellement la mémoire lors de l'exécution du programme, c'est-à-dire pour réserver des espaces en mémoire, désigner ces espaces et les rendre lorsqu'ils ne sont plus utilisés. On parle alors d'allocation dynamique (contrôlée).

Les *variables dynamiques* vont nous permettre :

- de prendre la place en mémoire au fur et à mesure des besoins, celle-ci étant bien sûr limitée par la taille de la mémoire,
- de libérer de la place lorsqu'on n'en a plus besoin.

Un classeur dans lequel on peut ajouter ou retirer des pages est une bonne image d'une variable dynamique

On accède à une variable dynamique grâce à un pointeur.

b- Les pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*.

Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable, qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets).

Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire, on parle alors de variable statique. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse, c'est une variable dynamique que nous allons définir ci-dessous.

Une adresse est une valeur que l'on peut stocker dans une variable. Les pointeurs sont justement des variables qui contiennent l'adresse d'autres variables d'un type donné.

Si un pointeur P contient l'adresse d'une variable A, on dit que '**P pointe sur A**'.

c- Les opérations de base :

La manipulation des pointeurs nécessite deux opérateurs :

- d'un opérateur 'adresse de': @ pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de': ^ pour accéder au contenu d'une adresse.

Déclaration :

<Nom du Pointeur>: ^ <typeObjet>;

Exemple : *p : ^entier ; « p est une variable de type pointeur vers un entier »*

*// en C on déclare int *p ;*

p est une **variable statique** de type pointeur vers un objet de type *entier*.

Représentation schématique

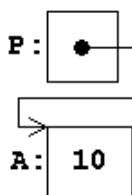
Soit **P** un pointeur sur un entier non initialisé, et **A** une variable (de type entier) contenant la valeur 10 :



L'opérateur 'adresse de' : @

<nomPointeur> @ (variable statique)

L'instruction **P @A;** affecte l'adresse de la variable A à la variable P. Dans la représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



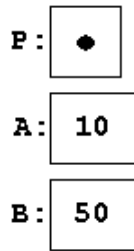
*//En langage C l'opérateur 'adresse de' utilisé est le symbole & alors on écrit :
p=&A*

L'opérateur 'contenu de' : ^

<NomPointeur>^ : désigne le contenu de l'adresse référencée (pointée) par le pointeur *<NomPointeur>*

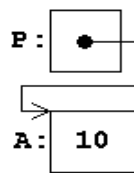
*//En langage C l'opérateur 'contenu de' utilisé est le symbole * alors on écrit :
p=10

Exemple : Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur:

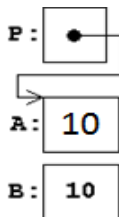


Après les instructions :

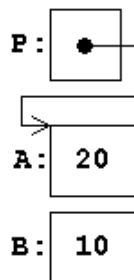
P @A; P pointe sur A



B P^; le contenu de A (Référéncé par P^) est affecté à B



P^ 20; le contenu de A (Référéncé par P^) est mis à 20.



d- Allocation dynamique de la mémoire:

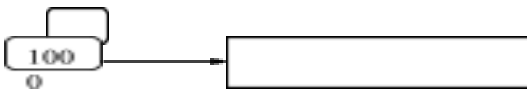
- Allocation d'une variable dynamique de **type simple**:

nom-pointeur : ^ <typeObjet>;

Allouer (nom-pointeur)

«Réservation d'une zone mémoire correspondant à une variable de type <typeObjet> et affecte au pointeur l'adresse de cette zone.»

Exemple 1:

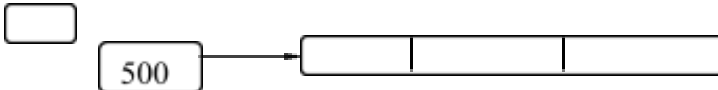
$p : ^{\text{entier}} ;$ p 
 $\text{Allouer}(p) ; p$
*// en langage C on écrit $p = (\text{int} *)\text{malloc}(\text{sizeof}(\text{int})) ;$*

Exemple 2 :

Type $\text{date} = \text{enregistrement}$

$\text{jour, mois, année} : \text{entier};$

$\text{finenreg};$

$d : ^{\text{date}} ;$ d 
 $d \text{ Allouer}(d)$
 *$d = (\text{date} *)\text{malloc}(\text{sizeof}(\text{date})) ;$*

L'accès à un champ de l'enregistrement : $\text{nom-pointeur}^{\text{.nom-champ}}$

Exemple : Si $(d^{\text{.mois}}=02)$ alors ...

$d1 : \text{date} ; d1.\text{jour} <- 22 ; d1.\text{mois} <- 01 ; d1.\text{annee} <- 2021$

e- Libérer la variable dynamique (pointée)

Libérer (<nom du pointeur>)

« Rendre disponible l'espace mémoire occupé par la variable dynamique créé par allouer »

Exemple:

variable $p : ^{\text{entier}} ;$

Allouer (p);

Libérer (p) ; *// en langage C : $\text{free}(p)$;*

f- Opérations courantes sur les pointeurs

- Affectation : on peut affecter un pointeur à un autre.

Exemple : $p1, p2 : ^{\text{entier}} ; A : \text{entier} ;$

$p1 @A ;$

$p2 p1 ;$

- Addition et soustraction :

Si P pointe sur l'élément $A[i]$ d'un tableau, alors

$P+n$ pointe $A[i+n]$

$P-n$ pointe sur $A[i-n]$

Exemple : $A : \text{tableau}[10] \text{ entier} ; p : ^{\text{entier}} ;$

$p A ;$ // ou bien $p = \&A[1]$

$p p+9$ // p pointe sur $A[10]$

$p p-1$ // p pointe sur $A[9]$

Constante nil

Le type pointeur admet une seule constante prédéfinie nil (**NULL en C**).

« $p \text{ nil}$ signifie que p pointe vers rien »

Exemple : p : ^entier ; p nil ;

Exemple :

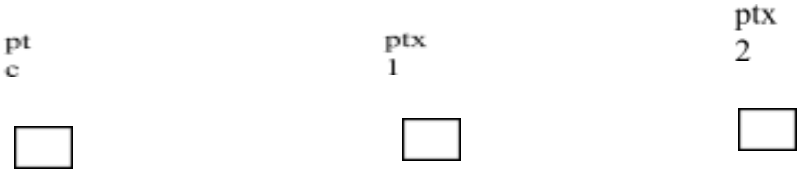
Variables

ptc : ^ch[20]char ;
ptx1, ptx2 : ^entier ;

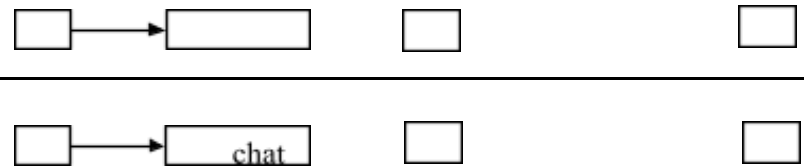
ptc<- nil ;
ptx1<- nil, ptx2<-nil ;

Début

Allouer(ptc) ;



ptc^ <- 'chat' ;



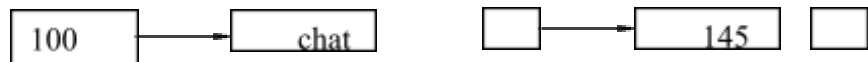
écrire (ptc^) ;

chat

Allouer(ptx1) ;

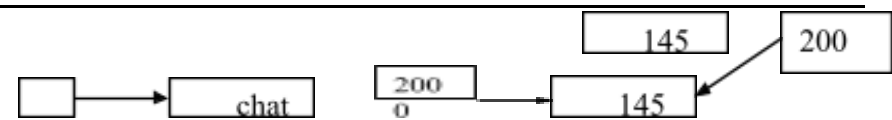


Lire(ptx1^) ;



Allouer(ptx2)

ptx2 <- ptx1 ;



écrire (ptx2^) ;

145

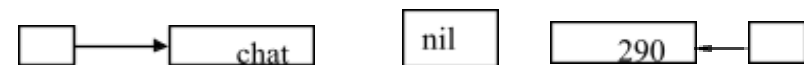
ptx2^ <- ptx1^ + ptx2^ ;



écrire (ptx1^, ptx2^) ;

290 290

ptx1 <- Nil ;



écrire (ptx2^) ;

290

Libérer(ptx2) ;



ptx2 <- nil ;

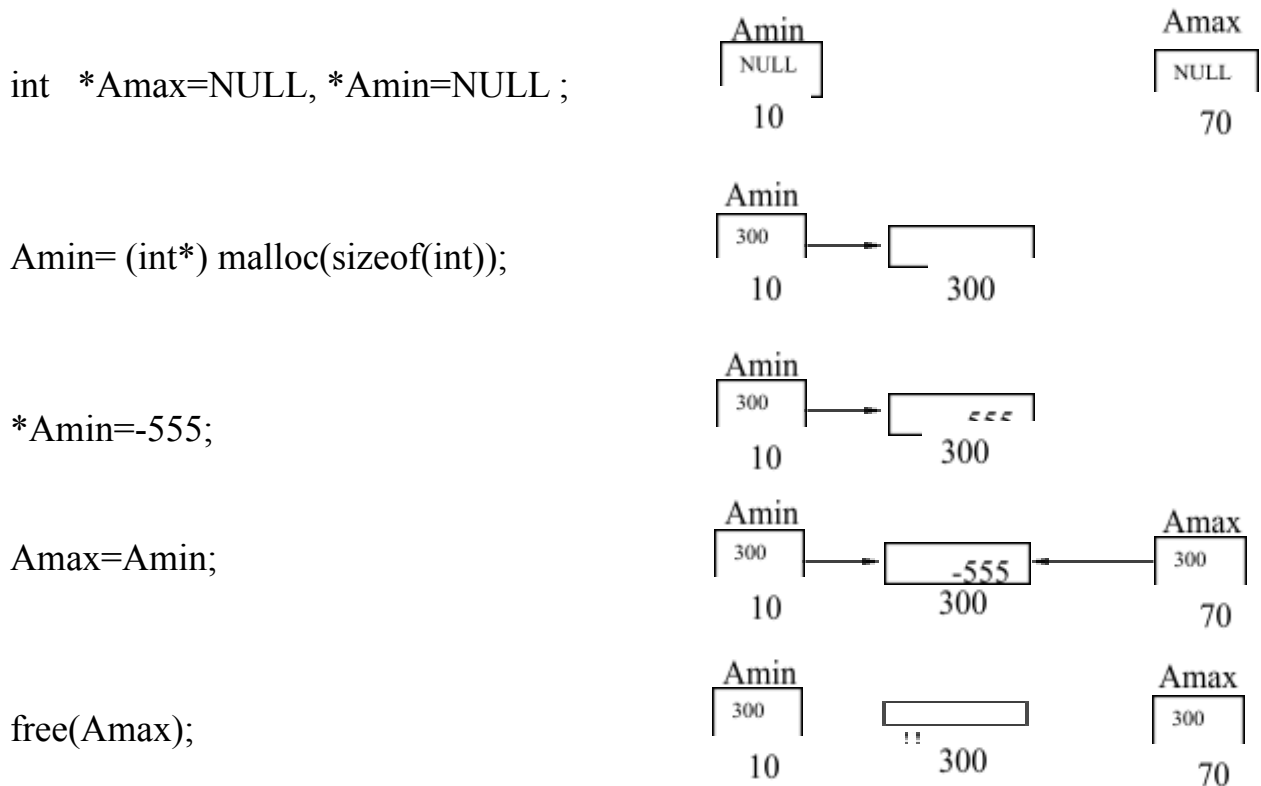


ptc <- nil ;



fin

Problème : on a coupé l'accès à la zone de mémoire occupée par la chaîne 'chat' sans la désallouer. Elle est irrécupérable



II.4 Listes chaînées

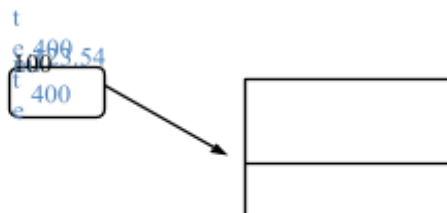
Une **liste chaînée** est une structure linéaire dynamique qui s'agrandit/diminue au fur et à mesure de l'ajout/suppression des données. Les éléments de cette liste, sont éparpillés dans la mémoire et reliés entre eux par des pointeurs. La liste est accessible uniquement par son premier élément (tête de liste).

Un **élément** d'une liste est une structure formée :

- d'une donnée ou information : entier, réel, caractère, chaîne, nouveau type,...
- d'un pointeur nommé *Suivant* indiquant la position (l'adresse) de l'élément le suivant dans la liste.

A chaque élément est associée une adresse mémoire.

Exemple :



La variable pointeur `tete` pointe sur l'espace mémoire `tete^` d'adresse 400. Cette cellule mémoire contient la valeur 123.54 dans le 1^{er} champ et la valeur spéciale *nil* dans le 2^{ème} champ. Ce dernier champ servira à indiquer quel est l'élément suivant de la liste. Dans notre cas, la valeur *nil* indique qu'il n'y a pas d'élément suivant.

a- Déclaration d'une liste chaînée

<p><u>Type</u> liste = enregistrement info :<typeElement> ; svt :^liste ; <u>finenreg</u> ; variable L :^liste ;</p>	<p><u>Type</u> liste=^Cellule ; <u>Type</u> Cellule=enregistrement info :<typeElement> ; svt :liste ; <u>finenreg</u> ; variable L,p : liste ;</p>
---	---

Remarque : on utilise la seconde représentation pour ne pas trainer avec nous le symbole : \wedge .

en Langage C

```
typedef struct ne *Liste ;
typedef struct ne
{ int info ;
  Liste svt ;
} noeud ;
```

b- **Création d'un nœud**

[illegible]

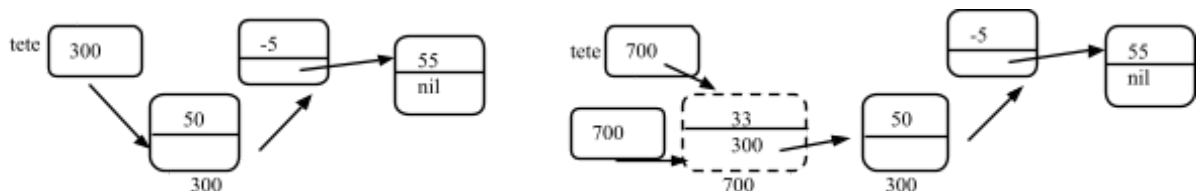
c- Accès à une valeur d'une liste

L, p : <u>Liste</u> ; entier : a ;	Liste L, p ; int a ;
---------------------------------------	-------------------------

<pre>a L^. info p<- L^.svt ;</pre>	<pre>a = L -> info ; p= L ->svt ;</pre>
--	---

d- Ajout d'un élément en tête de liste

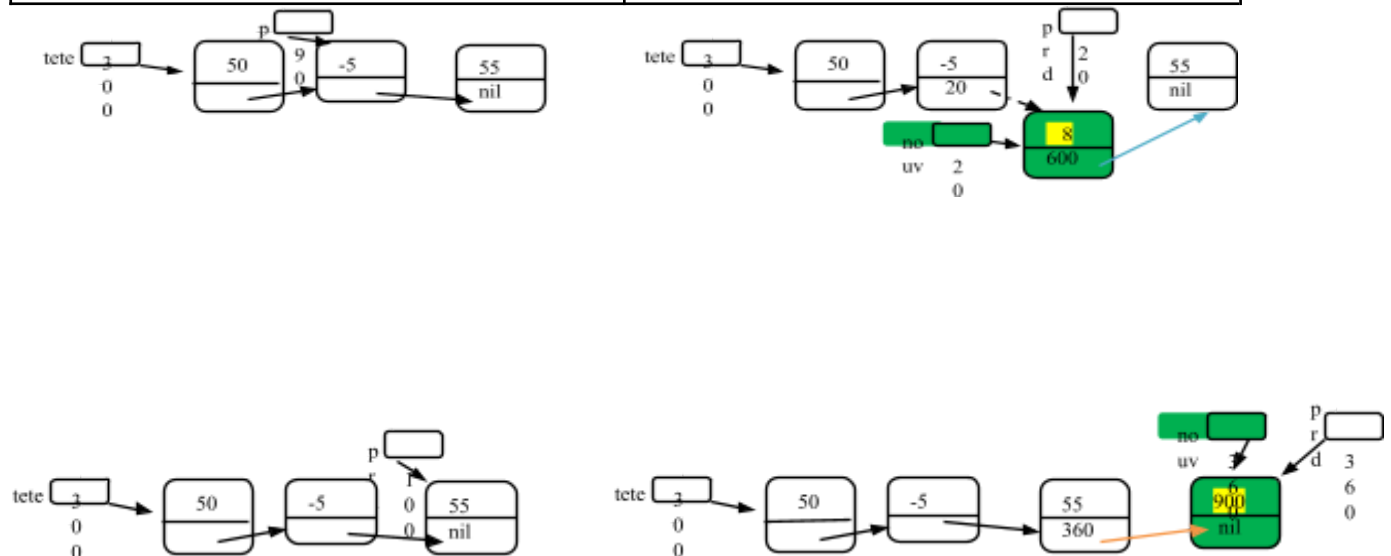
<p>Fonction ajout_tete(E/ tete : <u>Liste</u> ; E/ <u>e</u> :<typeElement>) : <u>Liste</u></p> <p>Variable nouv : <u>Liste</u> ;</p> <p>Début nouv creer_noeud () ;// <u>Allouer(nouv)</u> si (nouv ≠ nil) alors nouv^.info e ; <u>nouv^.svt tete ;</u> tete nouv ; sinon ecrire(« mémoire pleine ») fsi ; retourner (tete) ;</p> <p>Fin ;</p>	<pre>Liste ajout_tete(Liste tete, typeElement e) { Liste nouv=creer_noeud () ; if (nouv != NULL) { nouv-> info=e ; nouv->svt=tete ; tete=nouv ; } return tete ; }</pre>
---	--



e- Ajout d'un élément après une adresse donnée

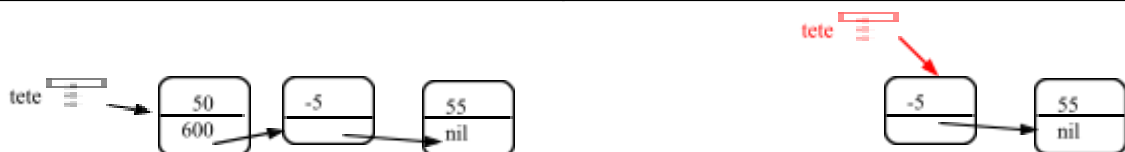
<p>Procédure ajout_après(ES/ prd : <u>Liste</u> ; E/ e : <typeElement>)</p> <p>Variable nouv : Liste ;</p> <p>Début nouv creer_noeud () ; si (nouv≠nil) alors <u>nouv^.info e ;</u> nouv^.svt prd^.svt ; prd^.svt nouv ; prd <- nouv ;</p>	<pre>void ajout_après(Liste *prd, typeElemet e) { Liste nouv=creer_noeud () ; if (nouv != NULL) { nouv->info=e ; nouv->svt=prd->svt ; prd->svt = nouv; *prd = nouv } }</pre>
---	---

<u>fsi ;</u>	<u>}</u>
<u>Fin ;</u>	



f- Suppression d'un élément en tête de liste

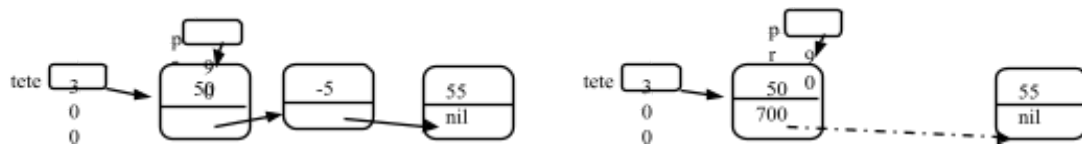
Fonction supprim_tete (E/ tete : <u>Liste</u>) : <u>Liste</u> Variable tmp : <u>Liste</u> ; Début tmp tete ; tete tete^.svt ; libérer (tmp) ; retourner (tete) ; Fin ;	Liste supprim_tete (Liste tete) { liste tmp ; tmp= tete ; tete = tete->svt ; free(tmp) ; return tete ; }
--	---



g- Suppression de l'élément après une adresse donnée

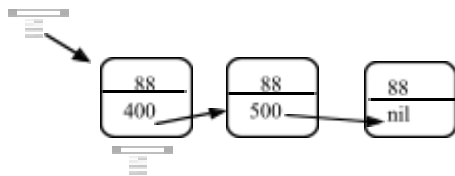
Procédure sup_Apre (E/ prd : <u>Liste</u>) Variable p : <u>Liste</u> ; Début p prd^.svt ; prd^.svt p^.svt ;	void sup_Apre (Liste prd) { Liste p ; p= prd -> svt ; prd -> svt = p -> svt ; free(p) ; }
---	--

libérer(p) ; Fin ;	}
------------------------------	---



h- Parcours d'une liste pour l'afficher

Procédure Affiche_liste(E/ tete : liste) Début Tantque (tete ≠ nil) faire Ecrire(tete^.info) ; tete = tete^.svt; fait ; Fin ;	void Affiche_liste (Liste tete) { while (tete != NULL) { Printf() ; tete = tete->svt; } }
--	--

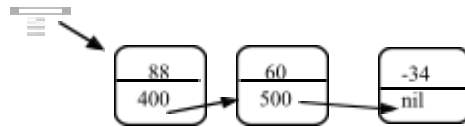


i- Recherche d'une valeur

Recherche d'une valeur donnée, dans une liste d'entiers **quelconques** et retourne son adresse, si elle existe sinon retourne **nil**

Fonction recherche(E/ tete : <u>Liste</u> , E/ val : <u>typeElem</u>) : <u>Liste</u> Début Tantque (tete ≠ nil et tete^.info ≠ val) Tantque (tete^.info ≠ val et tete ≠ nil) faire tete = tete^.svt; Fait ;	Liste recherche (Liste tete, int val) { while (tete != NULL && tete->info != val) { tete = tete->svt; } return tete ; }
--	---

retourner tete; Fin;	
--------------------------------	--



Val= -340

Ptr <- recherche(tete -34) ;

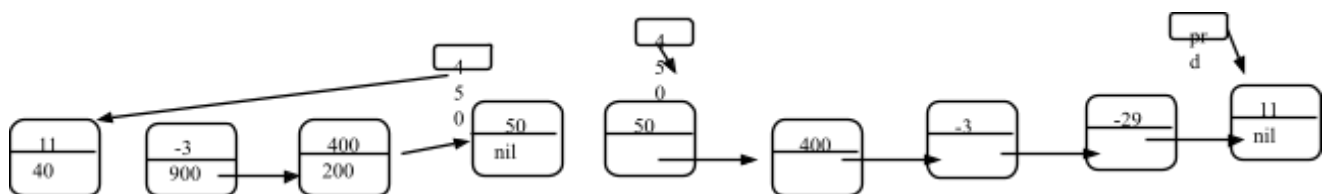
Recherche d'une valeur donnée, dans une liste d'entiers ***triés*** par ordre croissant et retourne l'adresse de l'élément précédent pour une éventuelle insertion (si la valeur n'existe pas) ou suppression (si la valeur existe).

Fonction rechercheT(E/ tete : <u>Liste</u> ; E/ val : entier) : <u>Liste</u> prd : <u>Liste</u> ; Début Prd <- nil Tantque (tete ≠ nil et tete^.info < val) faire prd tete; tete tete^.svt; fait ; retourner prd; Fin;	Liste rechercheT(<u>Liste</u> tete , int val) { Liste prd = NULL; while (tete != NULL && tete->info < val) { Prd = tete ; tete = tete->svt; } return prd; }
--	---

Création d'une liste :

Type liste=^Cellule ; Type Cellule=eng info :<typeElement> ; svt : <u>liste</u> ; feng ; variable tete, p: <u>liste</u> ; x : <u>entier</u> ; Debut tete <- nil Lire (x) ;	Type liste=^Cellule ; Type Cellule=eng info :<typeElement> ; svt : <u>liste</u> ; freg ; variable tete, p, prd: <u>liste</u> ; x : <u>entier</u> ; Debut tete <- nil Lire (x) ;
---	--

<p><u>Tantque</u> (x <> -1)</p> <p><u>Faire</u></p> <p>tete <- ajout_tete (tete, x) ;</p> <p>Lire (x) ;</p> <p><u>Fait</u> ;</p> <p>Affiche_liste(tete) ;</p> <p><u>Fin</u></p>	<p><u>si</u> x <> -1 <u>alors</u></p> <p>tete <- ajout_tete (tete, x) ;</p> <p>prd <- tete ;</p> <p>Lire (x) ;</p> <p><u>Tantque</u> (x != -1)</p> <p><u>Faire</u></p> <p>ajout_après (prd, x) ;</p> <p>Lire (x) ;</p> <p><u>fsi</u></p> <p><u>Fait</u> ;</p> <p><u>Fsi</u></p> <p>Affiche_liste(tete) ;</p> <p><u>Fin</u></p>
--	--



Remarque : pour x= {50, 400, -3, -29, 11, -1} :

- le premier algorithme affiche la liste suivante : 11, -29, -3, 400, 50. Puisque on insère toujours en tête de liste, le dernier élément inséré sera toujours en tête de liste. On parle alors de création en LIFO : Last In First Out.
- le deuxième algorithme affiche la liste suivante : 50, 400, -3, -29, 11. On ajoute toujours des éléments en fin de liste. On parle de création en FIFO : First In First Out.