

CHAPITRE II

Structures de Données Élémentaires

Structures de Données Élémentaires

Les structures de données présentées dans ce chapitre, constitueront un bagage important et nécessaire pour la construction d'algorithmes. Un choix judicieux d'une structure de données pour un algorithme, permet d'obtenir une complexité réduite. Plusieurs structures de données sont souvent possibles pour implémenter un algorithme. Le choix de l'algorithme à implémenter serait celui qui présente les meilleures complexités temporelle et/ou spatiale. Les structures de listes, queues, piles et ensembles sont abordées et les exercices proposés à la fin du chapitre renforceront la maîtrise de la manipulation de ces structures.

Listes chainées

La liste est un concept mathématique défini comme étant un ensemble d'éléments ordonnés. Si $L = \{a_1, a_2, \dots, a_n\}$ est une liste contenant les éléments a_i pour i allant de 1 à n , la notion d'ordre est très importante et elle signifie que l'élément a_{i+1} doit suivre l'élément a_i quelque soit i allant de 1 à $n-1$.

Le concept de liste en informatique est très important en algorithmique et permet de développer des algorithmes très souples pour des problèmes concrets de la vie courante. Un exemple est la représentation d'une queue ou file d'attente, concept utilisé couramment lorsqu'une ressource est partagée entre plusieurs utilisateurs humains ou processus informatiques.

La représentation d'une liste en informatique et plus précisément en mémoire peut se faire à l'aide d'un tableau ou d'une structure dynamique. Dans le cas où le langage de programmation permet une allocation dynamique de la mémoire, les deux structures peuvent être utilisées. Par contre pour les langages qui ne permettent pas d'allocation dynamique de la mémoire, seule la structure de tableau peut convenir et être exploitée. Pour un langage qui offre la possibilité d'utilisation des deux structures, le choix de l'une ou de l'autre structure est étroitement lié à la nature du problème à résoudre et non au langage de programmation car chacune des deux structures présentent des avantages et des inconvénients.

Les opérations de recherche, d'insertion et de suppression d'un élément seront considérées dans cette section et développées dans le détail dans le corrigé des exercices couvrant ce chapitre.

Représentation à l'aide de tableau et opérations élémentaires

La liste $L = \{a_1, a_2, \dots, a_n\}$ est représentée à l'aide des tableaux élément et suivant comme suit :

	élément	suivant
tête=1	a_1	2
2		3

n	a_n	-1
libre		
max		

Le tableau *élément* contient les éléments de la liste alors que le second *suivant* contient le chainage qui spécifie l'ordre selon lequel les éléments se suivent dans la liste. La valeur -1 indique que l'élément correspondant n'a pas de successeur, il est donc le dernier de la liste. Une liste chaînée est spécifiée à l'aide de l'indice du premier élément. Dans notre cas cet indice est nommé 'tête'. Avec cet indicateur, les éléments d'une liste peuvent être connus à tout moment grâce au chainage qui définit l'ordre d'apparition des éléments dans la liste. Par ailleurs, une partie des deux tableaux est occupée par les éléments de la liste et l'autre est libre permettant l'insertion d'autres éléments dans la liste. L'indice *libre* permet d'accéder à la zone libre du tableau. Toutes les cases vides sont accessibles à partir de *libre* et grâce au chainage. Tout compte fait, deux listes existent dans le tableau : *tête* qui permet de connaître les éléments de la liste et *libre* qui permet d'accéder à l'espace libre.

Pour rechercher un élément dans la liste, la manière la plus simple est de parcourir le tableau *élément* selon l'ordre indiqué par le vecteur *suivant* et de comparer à chaque fois l'élément recherché avec l'élément

courant du tableau. Si la liste n'est pas triée, l'algorithme de recherche de l'élément x doit parcourir toute la liste. Dans ce cas la complexité de l'algorithme est égale à la longueur de la liste et donc au pire elle est égale à $O(n)$ lorsque la liste est pleine. Si la liste est triée par ordre croissant par exemple, la recherche de l'élément x se fait uniquement sur la première partie de la liste où les éléments de la liste sont plus petits que x . Au pire cas, la complexité est en $O(n)$ comme pour le cas d'une liste non triée. Cependant la différence réside dans la complexité moyenne où elle se confond avec la complexité au pire dans le cas d'une liste non triée et elle est plus faible que la complexité au pire dans le cas d'une liste triée.

Si les éléments sont triés, l'insertion d'un élément consiste à placer l'élément dans la liste à la position qui convient par respect à l'ordre préétabli. Nous supposons que x n'appartient pas à la liste auparavant. Il ne faut pas oublier dans ce cas de gérer l'espace libre. L'action à entreprendre est de restreindre l'espace libre en manipulant l'indice libre.

La suppression d'un élément consiste à rechercher d'abord l'élément qu'on veut supprimer s'il existe dans la liste, ensuite à procéder à la suppression proprement dite. Il faut penser aussi à restituer l'espace libéré à la liste *libre*.

L'initialisation des listes *tête* et *libre*, les procédures *liste-vide* et *liste-pleine*, les algorithmes de recherche, d'insertion et de suppression sont décrits comme suit :

Initialisation des listes tête et libre ;

(initialement la liste tête est vide et la liste libre est pleine *)*

élément, suivant : tableau [1..max] d'entier ;

var tête, libre, p :-1..max

début

pour p := 1 à max-faire

début élément[p] := blanc ;

suivant[p] := p+1 ;

fin ;

élément [max] := blanc ; suivant[max] := -1

tête := -1 ;

libre := 1 ;

fin

procédure liste-vide ;
entrée : tête, libre : -1..max ;
sortie : vrai ou faux ;

début

si tête = -1 alors retourner (vrai) sinon retourner (faux) ;

fin

procédure liste-pleine ;
entrée : tête, libre : -1..max ;
sortie : vrai ou faux ;

début

si libre = -1 alors retourner(vrai) sinon retourner(faux) ;

fin

procédure recherche ;
(* recherche d'un entier x dans une liste non triée *)
entrée : tête : -1..max ; x : entier ;
sortie : position de x dans la liste ;

var p : -1..max ;

début

si (non liste-vide) alors

début p := tête ;

tant que (élément[p] ≠ x) et (suivant[p] ≠ -1) faire

p := suivant[p] ;

si (élément[p] = x) alors retourner(p)

sinon signaler ('x n'existe pas dans la liste') ;

fin

sinon signaler ('liste vide') ;

fin

procédure suppression ;
(* suppression d'un entier x d'une liste non triée *)
entrée : tête, libre : -1..max ; x : entier ;
sortie : tête, libre : -1..max ;
var p, q : -1..max ;

début
 si (*non liste-vide*) *alors*
 début $p := \text{tête}$;
 tant que (*élément*[p] $\neq x$) *et* (*suivant*[p] $\neq -1$) *faire*
 début $q := p$;
 $p := \text{suivant}[p]$;
 fin ;
 si (*élément*[p] $= x$) *alors*
 début si ($p = \text{tête}$) *alors* $\text{tête} := \text{suivant}[p]$
 sinon $\text{suivant}[q] := \text{suivant}[p]$;
 $\text{suivant}[p] := \text{libre}$;
 $\text{libre} := p$;
 fin
 fin
 sinon signaler ('*liste vide*') ;
fin

procédure insertion ;
(* *insertion d'un entier x dans une liste triée* *)
entrée : $\text{tête}, \text{libre} : -1..max$; $x : \text{entier}$;
sortie : $\text{tête}, \text{libre} : -1..max$;

var $p, q, \text{temp} : -1..max$;
début

$p := \text{tête}$;
élément [*libre*] $:= x$;
temp $:= \text{suivant}[\text{libre}]$;
si (*non liste-vide*) *et* (*non liste-pleine*) *alors*
 début tant que (*élément*[p] $< x$) *et* (*suivant*[p] $\neq -1$) *faire*
 début $q := p$;
 $p := \text{suivant}[p]$;
 fin ;
 si (*élément*[p] $> x$) *alors*
 si ($p = \text{tête}$) *alors* *début* $\text{suivant}[\text{libre}] := \text{tête}$;
 $\text{tête} := \text{libre}$;
 fin
 sinon début $\text{suivant}[\text{libre}] := p$;
 $\text{suivant}[q] := \text{libre}$;
 fin

```

sinon début suivant[libre]:= suivant[p];
    suivant[p] := libre ;
```

fin :

```

    libre:=temp ;
```

fin

```

sinon si (liste-vide) alors début tête:= libre ;
    suivant[tête] := -1 ;
```

```

    libre:=temp ;
```

fin

```

sinon signaler ('liste pleine')
```

fin

Exemple 2.1

Considérer la liste triée suivante constituée de 5 nombres entiers relatifs ordonnés et représentée par les tableaux *élément* et *suivant*:

	élément	suivant
Tête=1	-32	2
2	0	3
3	12	4
4	78	5
5	341	-1
Libre=6		7
7		8
8		-1

L'insertion du nombre 45 engendre la configuration suivante de la liste et le contenu suivant des tableaux :

	élément	suivant
Tête=1	-32	2
2	0	3
3	12	6
4	78	5
5	341	-1
6	45	4
Libre=7		8
8		-1

La suppression de 78 de cette nouvelle configuration donne le tableau suivant :

	élément	suivant
Tête=1	-32	2
2	0	3
3	12	6
Libre=4	78	7
5	341	-1
6	45	5
7		8
8		-1

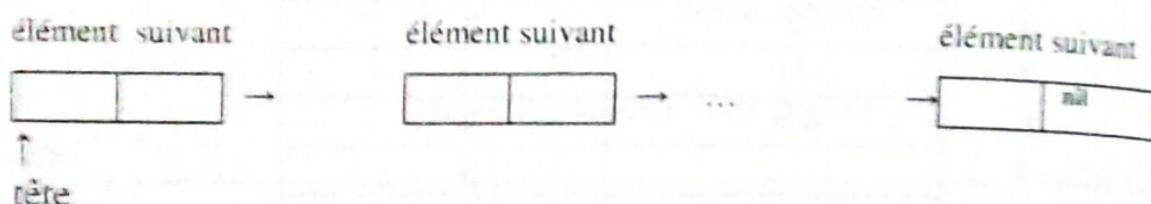
L'inconvénient de l'implémentation des listes avec des tableaux est dû à l'aspect statique de la structure. En effet, il faut prévoir une longueur du tableau suffisamment grande pour pouvoir exécuter toutes les opérations d'insertion à faire au niveau de la liste.

La complexité de chacune des trois procédures est $O(n)$ car au pire cas, on parcourt le tableau en entier pour :

- rechercher l'élément dans le cas de la première procédure,
- insérer l'élément en queue de liste pour la deuxième procédure ou
- supprimer un élément qui n'existe pas dans la liste pour la troisième procédure.

Représentation à l'aide de structure dynamique et opérations élémentaires

Tous les traitements vus dans la section précédente peuvent être conduits sur des structures de données dynamiques, en l'occurrence sur des listes dynamiques. La représentation graphique d'une liste dynamique est la suivante :



La liste dynamique utilise la structure d'enregistrement et la notion de pointeurs. Un enregistrement est constitué de deux ou plusieurs champs contenant l'ensemble de l'information de la liste. Dans notre cas, deux champs sont spécifiés : *élément* et *suivant*. Le champ *élément* contient un élément de la liste, en l'occurrence un entier et le champ *suivant* indique le chainage établi entre les éléments de la liste. A la différence de la structure statique du tableau, le chaînage est géré par le système, de même que l'espace requis pour les opérations d'insertion et l'espace libéré par les opérations de suppression.

La déclaration de la liste en algorithmique se fait de la manière suivante :

type

liste: *enregistrement élément : entier ; suivant : ↑liste ; fin* ;

var

tête : ↑liste ;

Notons que la fin de la liste est spécifiée à l'aide de la valeur *nil* affectée au champ *suivant* du dernier élément. D'autre part, la zone mémoire utilisée pour représenter la liste peut ne pas être contiguë. Sa taille ainsi que sa constitution peuvent être modifiées pendant l'exécution du programme. Elle est difficile à connaître pendant la compilation du programme, ce qui lui confère le statut de dynamique.

Le principe des opérations de recherche, d'insertion ou de suppression d'un élément d'une liste dynamique est exactement le même que celui qui s'effectue sur des listes représentées par des tableaux. La seule différence réside au niveau du mode de manipulation de la structure de données utilisée. Par exemple, pour rechercher un élément dans la liste dynamique, il suffit de parcourir la liste *tête* selon l'ordre indiqué par le champ de l'enregistrement *suivant* et de comparer à chaque fois l'élément recherché à l'élément contenu dans le champ de l'enregistrement *élément*.

Si les éléments sont triés, l'insertion d'un élément consiste à placer l'élément dans la liste à la position qui convient par respect à l'ordre préétabli. Nous supposons que x n'appartient pas à la liste auparavant.

La suppression d'un élément consiste à rechercher d'abord l'élément qu'on veut supprimer s'il existe dans la liste, ensuite à procéder à sa suppression. L'initialisation de la liste, les procédures *liste-vide* et *liste-pleine*, les algorithmes respectifs de recherche, d'insertion et de suppression sont les suivants :

Initialisation de la liste;

(* initialement la liste tête est vide *)

var tête : ↑liste ;

tête := nil ;

procédure liste-vide ;

entrée : tête : ↑liste ;

sortie : vrai ou faux ;

début si tête = nil alors retourner(vrai) sinon retourner (faux) ;

fin

procédure recherche ;(* rechercher un entier x dans une liste non triée *)

entrée : tête : ↑liste ; x : entier ;

sortie : position de x dans la liste ;

var $p : \uparrow\text{liste} ;$
début $p := \text{tête} ;$
 si(*liste-vide*) **alors signaler** ('*liste vide*')
 sinon début tant que ($p\uparrow.\text{élément} \neq x$) **et** ($p\uparrow.\text{suivant} \neq \text{nil}$) **faire**
 $p := p\uparrow.\text{suivant};$
 si($p\uparrow.\text{élément} = x$) **alors retourner**(p)
 sinon signaler('*x n'existe pas dans la liste*');
 fin
fin

procédure *insertion* ; (* *insertion de x dans une liste triée* *)

entrée : *tête* : $\uparrow\text{liste}$; *x* : *entier* ;

sortie : *tête* : $\uparrow\text{liste}$;

var $p, q, \text{libre} : \uparrow\text{liste};$
début $p := \text{tête} ;$
 nouveau (*libre*) ;
 $\text{libre}\uparrow.\text{élément} := x ;$
 si (*liste-vide*) **alors début tête** := *libre* ;
 $\text{libre}\uparrow.\text{suivant} := \text{nil} ;$
 fin
 sinon début tant que ($p\uparrow.\text{élément} < x$) **et** ($p\uparrow.\text{suivant} \neq \text{nil}$) **faire**
 début $q := p ;$
 $p := p\uparrow.\text{suivant};$
 fin ;
 si ($p\uparrow.\text{élément} > x$) **alors début** $\text{libre}\uparrow.\text{suivant} := p ;$
 $q\uparrow.\text{suivant} := \text{libre} ;$
 fin
 sinon début $\text{libre}\uparrow.\text{suivant} := p\uparrow.\text{suivant} ;$
 $p\uparrow.\text{suivant} := \text{libre} ;$
 fin
 fin
fin

procédure suppression ;
(suppression de x d'une liste non triée *)*
entrée : tête : \uparrow liste ; x : entier ;
sortie : tête : \uparrow liste ;

var p, q : \uparrow liste ;
début

si(liste-vide) alors signaler ('liste vide')
sinon début

p := tête ;
tant que ($p\uparrow.\text{élément} \neq x$) et ($p\uparrow.\text{suivant} \neq \text{nil}$) faire

début q := p ;
p := $p\uparrow.\text{suivant}$;

fin ;
si ($p\uparrow.\text{élément} = x$) alors

début si p = tête alors tête := $p\uparrow.\text{suivant}$
sinon $q\uparrow.\text{suivant} := p\uparrow.\text{suivant}$;
libérer(p) ;

fin ;
sinon signaler ('x n'existe pas dans la liste') ;

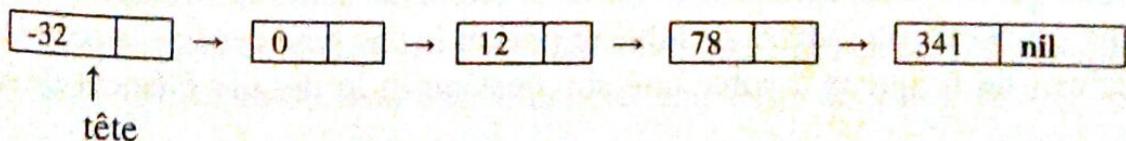
fin

fin

Une différence notable entre la structure statique de tableau et la structure dynamique de liste réside au niveau de l'allocation mémoire. Dans le cas du tableau, l'allocation de la mémoire est gérée par le programmeur. La taille du tableau est déclarée dans le programme et le débordement est contrôlé également par le programmeur pour éviter les effets de bord. Dans le cas des structures dynamiques, l'allocation se fait par le système à chaque fois que le programme nécessite de la mémoire pour construire la liste.

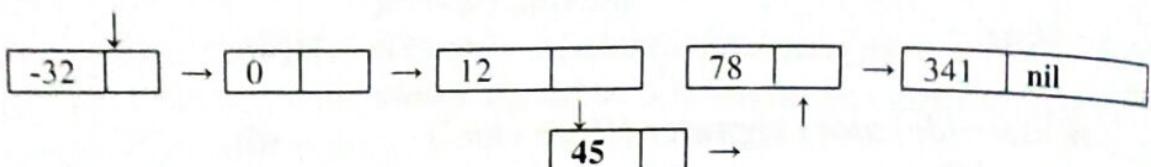
Exemple 2.2

Si l'on considère la liste des 5 entiers {-32, 0, 12, 78, 341}, la structure schématique de la liste dynamique contenant ces entiers est la suivante :



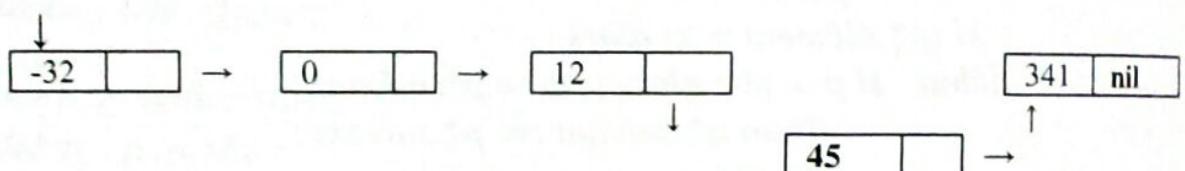
L'insertion du nombre 45 engendre la configuration suivante de la liste :

tête



La suppression de 78 de cette nouvelle configuration donne la configuration suivante :

tête



La complexité de chacune des procédures présentées est $O(n)$ pour la même raison évoquée dans l'utilisation des tableaux statiques. Au pire cas, la liste est parcourue dans sa globalité et les n éléments sont visités.

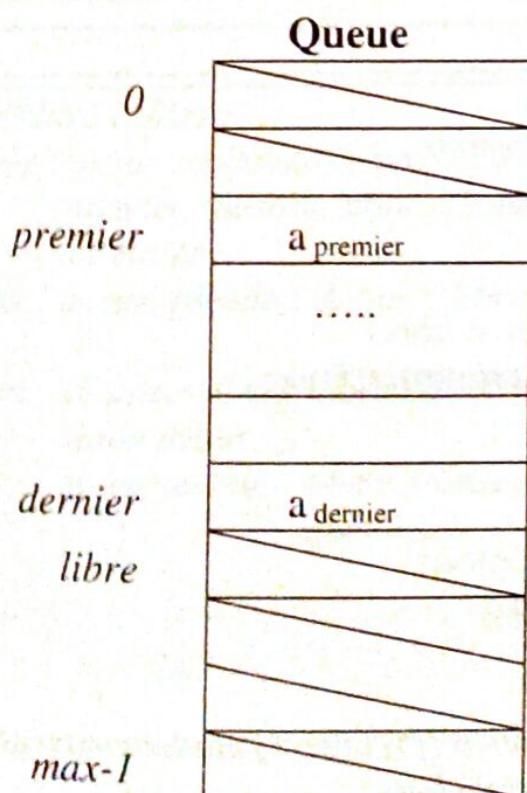
Queues ou files d'attente

Le concept de queue ou file d'attente est utilisé lorsque l'on partage une ressource entre plusieurs utilisateurs ou processus. L'affectation de la ressource généralement repose sur la stratégie dite FIFO (First In First Out), qui signifie que le premier qui a demandé l'exploitation de la ressource sera le premier servi. En informatique, ce concept est très bien modélisé et utilisé dans le partage des ressources informatiques internes telles que des pages ou segments de la mémoire et les périphériques d'entrée/sortie. Une queue est une liste particulière dédiée pour répondre à la stratégie FIFO, qui consiste à insérer un nouvel élément en fin de queue et à retirer le premier élément de la queue. Elle peut donc utiliser soit des tableaux ou des listes dynamiques pour sa représentation. Afin de mieux gérer les insertions et le retrait d'éléments selon la stratégie FIFO, deux pointeurs ou indices de tableaux selon le cas pour repérer le premier élément de la queue à subir une suppression et le dernier élément de la

queue afin de pouvoir insérer le prochain élément qui rentre dans la queue après ce dernier élément, sont nécessaires.

Représentation à l'aide de tableau et opérations élémentaires

Une queue peut être représentée par un tableau et deux indices caractérisant les positions du *premier* et du *dernier* élément. Ici, nous n'avons pas besoin de chainage car l'insertion se fait à la fin de la queue et l'extraction d'un élément s'effectue du début de la queue. Nous supposons que les éléments se trouvent dans un espace contigu et qu'aucune opération d'insertion (et de suppression) ne soit autorisée en dehors de celle s'effectuant en fin de la queue (en début de la queue). Le tableau suivant appelé *Queue* peut contenir jusqu'à *max* éléments. Il peut être vu comme un tableau circulaire dans le sens où la partie remplie s'étend de *premier* jusqu'à *dernier* et la partie vide de *libre* jusqu'à *premier -1*. Les cases hachurées correspondent à des cases vides.



L'opération de recherche d'un élément dans la queue est plus simple que celle des listes. L'insertion d'un élément dans la queue se fait en

appelant la procédure *enfiler* décrite ci-dessous. Elle est beaucoup plus simple car elle se fait directement après *dernier*. Il faut juste bien traiter le cas où la queue est vide, auquel cas il faut réinitialiser *premier*. Dans le cas où la queue est pleine, il faut signaler le débordement. Si ce dernier se présente très fréquemment, il faut penser à augmenter la taille du tableau.

La suppression d'un élément est effectuée par la procédure *défiler*. Elle est également simple car elle consiste à retirer directement le premier de la queue. Là aussi, il faut signaler le cas où l'opération est impossible lorsque la queue est vide.

L'initialisation de la queue, les algorithmes de recherche, d'insertion, de suppression ainsi que la procédure de test de l'état de la queue sont les suivants :

Initialisation de la queue :

max := 200 ; premier := 0 ; dernier := 0 ; libre := 0 ;

procédure queue-vide ;

entrée : queue : tableau[0..max-1] d'entier ;

premier, dernier, libre : 0..max-1 ;

sortie : vrai ou faux ;

début si(premier = libre) et (dernier = libre)

alors retourner(vrai) sinon retourner(faux) ;

fin

procédure queue-pleine ;

entrée : queue : tableau[0..max-1] d'entier ;

premier, dernier, libre : 0..max-1 ;

sortie : vrai ou faux ;

début si (libre = premier) et (dernier = (premier-1) modulo max) alors

retourner(vrai) sinon retourner(faux) ;

fin

procédure recherche ;

entrée : queue : tableau[0..max-1] d'entier ;

```
premier, dernier : 0..max-1 ;
x : entier ;
sortie : position de x dans la queue ;

var    p : entier ;
trouve : booléen ;

début si (non queue-vide) alors
    début   trouve := faux ;
        p := premier ;
        tant que (non trouve) et (p <= dernier) faire
            si (queue[p] = x) alors trouve := vrai
        sinon p := (p+1) modulo max ;
            si (trouve) alors retourner (p)
            sinon signaler ('x n'existe pas dans la queue') ;
    fin sinon signaler ('queue vide')
fin
```

```
procédure enfiler ;
entrée : queue : tableau[0..max-1] d'entier ;
        premier, dernier, libre : 0..max-1 ;
        x : entier ;
sortie : queue, premier, dernier, libre;

début si queue-pleine alors signaler ('queue pleine')
    sinon début
        si queue-vide alors premier := libre ;
            queue [libre] := x ;
            dernier := libre ;
            libre := (libre + 1)modulo max ;
    fin
```

```
procédure défiler;
entrée : queue : tableau[0..max-1] d'entier ;
        premier, dernier, libre : 0..max-1 ;
```

sortie : x, le premier élément de la queue ;

var x ;
début *si (queue-vide) alors signaler ('queue vide')*
 sinon début x := queue [premier] ;
 premier := (premier +1) modulo max ;
 si (premier = libre) alors dernier := libre ;
 retourner (x) ;
 fin
fin

Exemple 2.3

Considérer la queue suivante constituée de 5 nombres entiers :

Queue	
premier= 0	-32
1	0
2	12
3	78
dernier= 4	341
libre=5	

L'insertion du nombre 45 engendre la configuration suivante de la queue :

Queue	
libre = premier=0	-32
1	0
2	12
3	78
4	341
dernier= 5	45

La suppression du premier élément de cette nouvelle configuration donne le tableau suivant :

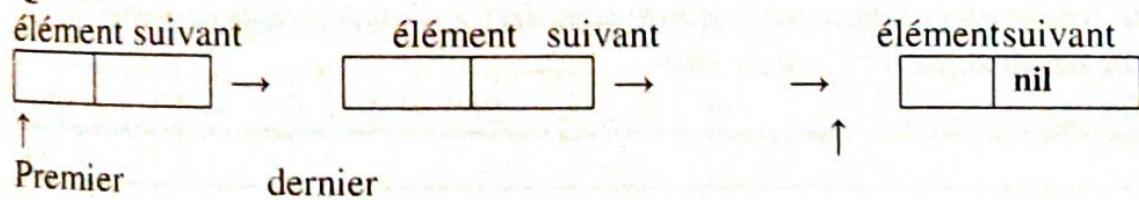
Queue

libre = 0	-32
premier = 1	0
2	12
3	78
4	341
dernier = 5	45

Représentation à l'aide de structure dynamique et opérations élémentaires

Les structures dynamiques peuvent être également utilisées pour représenter des queues. La figure suivante est une représentation schématique d'une queue à l'aide d'une liste dynamique.

Queue



La déclaration du type de queue est :

queue : enregistrement premier, dernier : ↑liste fin ;

Les techniques de recherche, d'insertion et de suppression d'un élément d'une queue en utilisant une structure dynamique sont analogues à celles utilisant des tableaux. L'initialisation d'une queue, la procédure queue-vide, les algorithmes de recherche, d'insertion et de suppression sont les suivants :

Initialisation de la queue :

queue.premier := nil ; queue.dernier := nil ;

procédure queue-vide ;

entrée : queue : enregistrement premier, dernier : \uparrow liste fin ;

sortie : vrai ou faux ;

début si (queue.premier = nil) et (queue.dernier = nil) **alors**
retourner(vrai)

sinon retourner(faux) ;

fin

procédure recherche ;

entrée : queue : enregistrement premier, dernier : \uparrow liste fin ; x : entier ;

sortie : position de x dans la queue ;

var p : \uparrow liste ; trouve : booléen ;

début si (non queue-vide) **alors**

début p := queue.premier ;

trouve := faux ;

tant que (non trouve) et (p \neq nil) **faire**

si((p \uparrow .élément=x) **alors** trouve := vrai **sinon** p := p \uparrow .suivant ;

si trouve **alors** retourner (p)**sinon** signaler ('x n'est pas dans la queue') ;

fin **sinon** signaler ('queue vide')

fin

procédure enfiler ;

entrée : queue : enregistrement premier, dernier : \uparrow liste fin ; x : entier ;

sortie : queue : enregistrement premier, dernier : \uparrow liste fin ;

var p : \uparrow liste ;

début nouveau(p) ;

p \uparrow .élément := x

p \uparrow .suivant := nil ;

si(queue.dernier \neq nil) **alors** queue.dernier \uparrow .suivant := p
sinon **début** queue.premier := p ;

queue. dernier := p;

fin

fin

procédure dépiler ;

entrée : queue : enregistrement premier, dernier : ↑liste fin ;
sortie : le premier élément de la queue ;

var p: ↑liste ;

début si (non queue-vide) alors

début p := queue. premier ;

queue. premier := queue. premier↑. suivant ;

si (queue. premier = nil) alors queue. dernier := nil ;

retourner (p↑. élément) ;

libérer (p) ;

fin

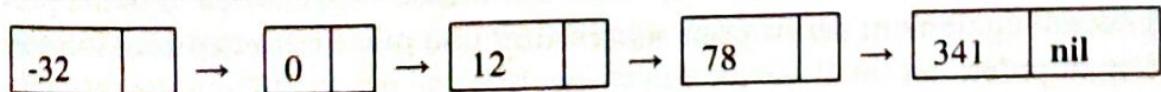
sinon signaler ('queue vide') ;

fin

Remarquons que la notion de *queue-pleine* n'existe pas pour les structures dynamiques car on peut rajouter autant d'éléments dans la queue que l'on désire. Comme pour le cas d'utilisation de structure statique, la complexité de chacune de ces trois procédures est $O(n)$.

Exemple 2.4

Considérer la liste suivante constituée des 5 nombres entiers relatifs:



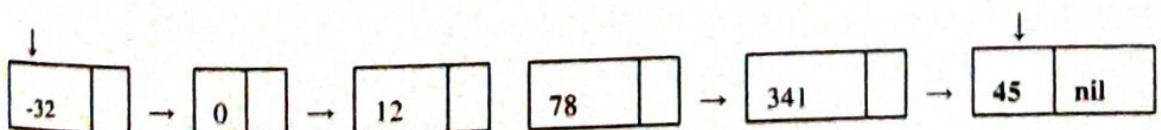
↑↑

Premier dernier

L'insertion du nombre 45 engendre la configuration suivante :

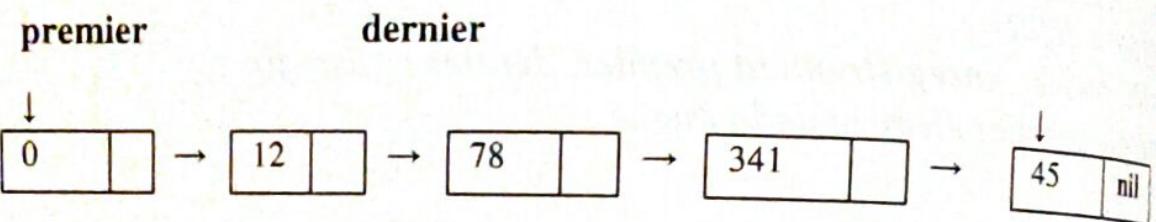
Premier

dernier



La suppression du premier élément de la queue de cette nouvelle configuration donne la configuration suivante :

$$x = -32$$



Piles

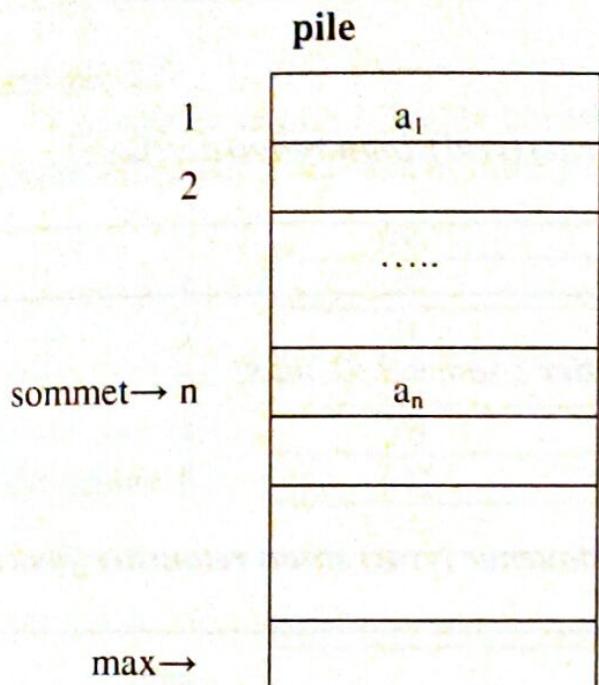
La pile est une structure de données gérée selon la stratégie FILO (First In Last Out) qui est équivalente à LIFO (Last In First Out). Elle est souvent présente dans le logiciel, elle sert en l'occurrence à l'implémentation de la récursivité et à la sauvegarde du contexte des processus en cours d'exécution en cas d'interruptions pour les systèmes d'exploitation.

De par la nature de la stratégie FILO, une pile sera représentée par une structure qui sera fermée d'un côté et ouverte de l'autre. Cette ouverture appelée *sommet de la pile* permettra d'insérer et de retirer des éléments de la pile. La structure la plus adéquate pour représenter une pile est le tableau. Dans le cas où l'élément de la pile se présente sous forme d'un n-up let, un tableau d'enregistrements dont les champs sont réservés pour stocker le n-up let, est utilisé. Les listes dynamiques peuvent également servir pour représenter une pile mais elles sont lourdes à manipuler.

Seules les opérations d'insertion et de suppression d'un élément ont un sens pour les piles.

Représentation à l'aide de tableau et opérations élémentaires

La pile contenant les éléments a_1, a_2, \dots, a_n , est représentée à l'aide du tableau suivant :



Ce tableau contient deux parties : la partie qui s'étend de 1 à *sommet* contient les éléments de la pile et la seconde partie libre allant de *sommet+1* à *max*, permet l'insertion d'autres éléments. Nous n'avons pas besoin de spécifier le pointeur *libre* car il se trouve à *sommet+1*. Bien entendu, comme la structure de tableau est statique, il est prudent à chaque manipulation de la pile de tester si la liste ne déborde pas (*sommet>max*). L'opération d'insertion d'un élément dans la pile est également appelée *empilement*, celle de la suppression *dépilement*. Ces deux opérations sont faciles à implémenter, voici les procédures d'écriture de ces deux opérations ainsi que la procédure testant si la pile est vide:

Initialisation de la pile

max := 200 ; sommet := 0 ;

procédure pile-vide

entrée : pile :tableau[1..max] d'entier ; sommet :0..max;

sortie : vrai ou faux ;

début

si(sommet = 0) alors retourner(vrai)sinon retourner(faux);

fin

procédure pile-pleine

entrée : pile :tableau[1..max] d'entier ; sommet :0..max;

sortie : vrai ou faux ;

début

si (sommet = max) alors retourner (vrai) sinon retourner(faux);

fin

procédure empiler ;

entrée : pile :tableau[1..max] d'entier ; sommet :0..max; x : entier ;

sortie : pile :tableau[1..max] d'entier ; sommet :0..max;

début si (non pile-pleine) alors

début sommet := sommet + 1 ;

pile[sommet] := x ;

fin

sinon signaler un débordement de la pile ;

fin

procédure dépiler;

entrée : pile :tableau[1..max] d'entier ; sommet :0..max;

sortie : l'élément qui se trouve au sommet de la pile ;

début si (non pile-vide) alors

début

```

    sommet := sommet - 1 ;
    retourner (pile[sommet+1]);
    fin
    sinon signaler que la pile est vide
fin

```

Exemple 2.5

Considérer la pile suivante constituée des 5 nombres entiers relatifs et représentée par le tableau nommé *pile* suivant:

pile	
1	-32
2	0
3	12
4	78
Sommet=5	341
max	

L'empilement du nombre 45 engendre la configuration suivante de la pile et le contenu suivant du tableau :

pile	
1	-32
2	0
3	12
4	78
5	341
sommet=6	45
max	

Le dépilement à partir de cette nouvelle configuration donne le tableau initial suivant :

pile	
1	-32
2	0
3	12
4	78
sommet= 5	341
	45
max	

$x = 45$

Ensembles

Un ensemble est un concept beaucoup utilisé dans les mathématiques modernes. Dans la vie courante, il est également très répandu. Sa modélisation informatique est donc nécessaire pour le traitement de problèmes manipulant des ensembles. De par sa définition, un ensemble contient des éléments qui ne sont pas supposés suivre un quelconque ordre. Nous pouvons imaginer plusieurs structures de données pour le modéliser.

Représentation et opérations élémentaires

Une représentation possible d'un ensemble S , qui facilite l'écriture des opérations union et intersection entre deux ensembles, utilise deux tableaux TV (Tableau des Valeurs) et VR (Vecteur Représentatif) comme l'illustre l'exemple suivant:

Exemple 2.6 :

Soit $S=\{9, 11, 546, 0, 27, 47\}$, un ensemble de 6 entiers naturels. Les tableaux TV et VR se présentent comme suit :

31	5	9	87	11	546	0	27	8	47	71	-1	-1		
1	2	3	4	5	6	7	8	9	10	11				

TV

0	0	1	0	1	1	1	1	0	1	0	0	0	0	6
1	2	3	4	5	6	7	8	9	10	11				max

VR

Le tableau TV contient toutes les valeurs susceptibles d'appartenir à un ensemble S. $TV[max]$ contient le nombre de valeurs de TV. VR, le vecteur représentatif de S, est un vecteur de nombres booléens défini comme suit :

$$VR[i] = \begin{cases} 1 & \text{si } TV[i] \in S \\ 0 & \text{sinon} \end{cases}$$

$VR[max]$ contient le nombre d'éléments de S. Les opérations qu'on peut mener sur les ensembles sont :

- Appartenance d'un élément à un ensemble.
- Union de deux ensembles.
- Intersection entre deux ensembles.

Ces trois opérations peuvent être implémentées comme suit :

Procédure appartenance ;

entrée : TV : tableau d'entiers ; VR : tableau de booléens ; x : entier ;

sortie : vrai ou faux ;

var p : entier ;

début p := 1;

tant que ($p \leq TV[max]$) et ($TV[p] \neq x$) faire $p := p + 1$;

si ($TV[p] = x$) et ($VR[p] = 1$) alors retourner(vrai)

sinon retourner(faux) ;

fin

procédure union ;

entrée : TV : tableau d'entiers ; VR₁, VR₂ : tableau de booléens ;

sortie : VR₃ : tableau de booléens ;

var i : 1 .. max ;

début

pour (i := 1 à TV[max]) faire

VR₃[i] := VR₁[i] ou VR₂[i] ;

(ou est l'opérateur booléen de disjonction *)*

fin ;

procédure intersection :

entrée : TV : tableau d'entiers ; VR₁, VR₂ : tableau de booléens ;
sortie : VR₃ : tableau de booléens ;

var i : 1 .. max ;

début

pour (i := 1 à TV[max]) faire

VR₃[i] := VR₁[i] et VR₂[i] ;

(et est l'opérateur booléen de conjonction *)*

fin ;

La complexité de la procédure appartenance est $O(TV[\max])$ car au pire cas, x n'appartient pas à TV et la boucle aura un nombre d'itérations égal à $TV[\max]$.

Les procédures *union* et *intersection* ont chacune une complexité en $O(TV[\max])$ car la boucle *pour*, a un nombre d'itérations égal à $TV[\max]$.

Exercices

Exercice 2.1

Considérer un tableau d'entiers trié par ordre croissant. Écrire une procédure de recherche d'un élément dans le tableau.

Exercice 2.2

En adoptant la représentation d'un ensemble vu dans ce chapitre, écrire les procédures suivantes :

- 1) Insertion d'un élément dans un ensemble.
- 2) Suppression d'un élément d'un ensemble.

Exercice 2.3

Considérer deux listes de nombres entiers triées par ordre croissant. On souhaiterait fusionner ces deux listes pour obtenir une troisième liste triée.

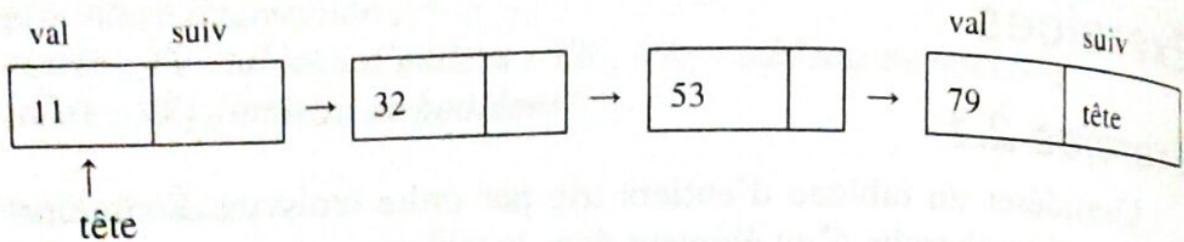
- 1) Écrire un algorithme de fusion de deux listes triées
 - a. En utilisant la structure de tableau.
 - b. En utilisant la structure dynamique.
- 2) Quelle est la complexité de votre algorithme ?
- 3) Comment représenter une liste chainée en assembleur ? Illustrer votre réponse à l'aide d'un petit exemple.
- 4) Écrire l'algorithme de fusion proposé en assembleur à l'aide du jeu d'instructions donné au premier chapitre. Calculer sa complexité.

Exercice 2.4

Écrire un algorithme pour créer une liste doublement chainée à partir d'une liste quelconque. Calculer sa complexité.

Exercice 2.5

Une liste chainée circulaire (en anneau) est une liste dont le dernier élément a pour successeur le premier élément de la liste. Un exemple d'une liste circulaire est le suivant :



- 1) Écrire un algorithme pour déterminer si une liste chainée est circulaire ou non. Calculer sa complexité.
- 2) Écrire un algorithme pour compter le nombre d'éléments d'une liste chainée circulaire.
- 3) De la liste ci-dessus, supprimer le nombre 79. Écrire un algorithme de suppression d'un entier d'une liste chainée circulaire d'entiers triée.
- 4) Insérer le nombre 96 dans la liste ci-dessus. Écrire un algorithme pour insérer un entier dans une liste chainée circulaire d'entiers triée.

Exercice 2.6 (liste miroir)

Considérer la liste de l'exercice 2.5.

- 1) Donner le schéma de la liste qui inverse l'ordre des éléments.
- 2) Écrire un algorithme pour inverser l'ordre d'une liste.
 - a. Donner une version itérative
 - b. Donner une version récursive
- 3) Calculer sa complexité dans les deux cas.

Exercice 2.7

Écrire une procédure pour concaténer deux listes simplement chainées.

Corrigé des exercices

Exercice 2.2

- 1) Écrire la procédure d'insertion d'un élément dans un ensemble et calculer sa complexité.

Procédure insertion ;
entrée : TV : tableau d'entiers ; VR : tableau de booléens ; x : entier ;
sortie : TV : tableau d'entiers ; VR : tableau de booléens ;
var p : entier ;
début $p := 1$;
 tant que ($p \leq TV[\max]$) et ($TV[p] \neq x$) **faire** $p := p + 1$;
 si ($TV[p] \neq x$) **alors début** $TV[p] := x$;
 $TV[\max] := TV[\max] + 1$;
 fin ;
 $VR[p] := 1$;
fin

La complexité de la procédure insertion est $O(TV[\max])$ car au pire cas, x n'appartient pas à TV et la boucle aura un nombre d'itérations égal à $TV[\max]$.

- 2) Écrire la procédure de suppression d'un élément d'un ensemble et calculer sa complexité.

Procédure suppression ;
entrée : TV : tableau d'entiers ; VR : tableau de booléens ; x : entier ;
sortie : TV : tableau d'entiers ; VR : tableau de booléens ;
var p : entier ;
début $p := 1$;
 tant que ($p \leq TV[\max]$) et ($TV[p] \neq x$) **faire** $p := p + 1$;
 si ($TV[p] = x$) **alors**
 si ($VR[p] = 0$) **alors afficher** ('x n'existe pas dans l'ensemble')
 sinon $VR[p] := 0$;
fin

La complexité de la procédure suppression est $O(TV[\max])$ car au pire cas, x n'appartient pas à TV et la boucle aura un nombre d'itérations égal à $TV[\max]$.

Exercice 2.3

- 1) L'algorithme de fusion de deux listes triées
 - a. En utilisant la structure de tableau

Chacune des deux listes est représentée par deux tableaux : élément et suivant. Soient tête1 et tête2 les indices respectifs des premiers éléments des listes. L'algorithme de fusion est comme suit :

```
algorithme fusion ;
entrée : tête1, tête2 : entier ;
sortie : tête3 : entier ;
const max = 200 ;
var p1, p2, p3, libre : entier ;
T1, T2, T3 : tableau [1..max] de
    enregistrement élément : entier ; suivant : 1..max fin;
début   p1 := tête1 ;
        p2 := tête2 ;
        libre := 1
        tant que (p1 ≠ -1) et (p2 ≠ -1) faire
            début   si (T1[p1].élément < T2[p2].élément) alors
                débutT3[libre].élément := T1[p1].élément ;
                    p1 := T1[p1].suivant ;
                    fin
                sinon débutT3[libre].élément := T2[p2].élément ;
                    p2 := T2[p2].suivant ;
                    fin ;
                T3[libre].suivant := libre+1 ;
                Libre := libre+1 ;
                fin ;
            si (p1 = -1) alors
                tant que (p2 ≠ -1) faire
                    débutT3[libre].élément := T2[p2].élément ;
                        T3[libre].suivant := libre+1 ;
                        Libre := libre+1 ;
                    fin sinon
                    tant que (p1 ≠ -1) faire
                        début T3[libre].élément := T1[p1].élément ;
                            T3[libre].suivant := libre+1 ;
                            Libre := libre+1 ;
                        fin ;
                    si (libre = 1) alors tête3 := -1 sinon début T3[libre-1].suivant := -1 ;
                                            tête3 := 1 ;
                    fin ;
fin ;
```

b. En utilisant la structure dynamique de listes

algorithme fusion :

entrée : tête1, tête2 : $\uparrow p$

sortie : tête3 : $\uparrow p$

var

p : enregistrement élément : entier ; suivant : $\uparrow p$ fin;
 $p1, p2, p3, libre, prec$: $\uparrow p$

début

$p1 := tête1$;

$p2 := tête2$;

si ($p1 \neq \text{nil}$) et ($p2 \neq \text{nil}$) alors

début nouveau(libre)

$tête3 := libre$

fin sinon $tête3 := \text{nil}$;

tant que ($p1 \neq \text{nil}$) et ($p2 \neq \text{nil}$) faire

début

si ($p1 \uparrow.\text{élément} < p2 \uparrow.\text{élément}$) alors

début $libre \uparrow.\text{élément} := p1 \uparrow.\text{élément}$;

$p1 := p1 \uparrow.\text{suivant}$;

fin

sinon début $libre \uparrow.\text{élément} := p2 \uparrow.\text{élément}$;

$p2 := p2 \uparrow.\text{suivant}$;

fin ;

$prec := libre$;

nouveau(libre) ;

$prec \uparrow.\text{suivant} := libre$;

fin ;

si ($p1 = \text{nil}$) alors

tant que ($p2 \neq \text{nil}$) faire

début $libre \uparrow.\text{élément} := p2 \uparrow.\text{élément}$;

$prec := libre$;

nouveau(libre) ;

$prec \uparrow.\text{suivant} := libre$;

fin

sinon tant que ($p1 \neq \text{nil}$) faire

début $libre \uparrow.\text{élément} := p1 \uparrow.\text{élément}$;

$prec := libre$;

nouveau(libre) ;
prec[↑].suivant := libre ;
fin ;
libérer(libre) ;
prec[↑].suivant := nil ;
fin ;

- 2) L'algorithme parcourt les deux listes fournies en entrée linéairement pour construire une troisième liste qui contient les éléments des deux listes. Si $l1$ et $l2$ sont les longueurs respectives des deux listes, la complexité de l'algorithme serait $O(l1+l2)$.
- 3) Pour représenter une liste chainée en assembleur, il faut réservé un espace mémoire contigu pour contenir les éléments de la liste en sachant qu'un élément est un enregistrement d'un entier et d'une adresse relative. Plus précisément, deux mots contigus seront nécessaires pour mettre l'entier et l'adresse de l'élément qui suit.

Exemple :

Soit $l = \{11, 12, 45, 14, 76\}$ une liste, sa représentation en assembleur est la suivante :

10	11
11	12
12	45
13	14
14	76
15	-1

- 4) Bien sûr, seule la représentation de tableau convient pour l'assembleur. L'algorithme de fusion en assembleur à l'aide du jeu d'instructions donné au chapitre 1 est comme suit:

MOV	tête1 , R1
MOV	tête2 , R2
MOV	tête3, R3
MOV	R3, R0 / libre := 1
ADD	#1, R1 tant que (p1 ≠ -1) et (p2 ≠ -1) faire
JZ	R1, L0 début

	ADD #1, R2	si ($T1[p1].élément < T2[p2].élément$) alors
	JZ R2, L0	début $T3[libre].élément := T1[p1].élément$;
	SUB #1, R1	$p1 := T1[p1].suivant$;
	SUB #1, R2	fin
	SUB (R2), (R1)	
	JGT (R1), L1	
	ADD (R2), (R1)	
	MOV (R1), (R3)	
	ADD #1, R1	
	MOV (R1), R1	
	JMP L2	
L1:	ADD (R2), (R1)	
	MOV (R2), (R3)	sinon début $T3[libre].élément := T2[p2].élément$;
	ADD #1, R2	$p2 := T2[p2].suivant$;
	MOV (R2), R2	fin ;
L2 :	ADD #2, R0	$T3[libre].suivant := libre+1$;
	ADD #1, R3	Libre := libre+1 ;
	MOV R0, (R3)	fin ;
	ADD #1, R3	
L0 :	JZ (R1), L4	
L6 :	JZ (R2), L3	
	JMP L5	si ($p1 = -1$) alors
L3 :	SUB #1, (R2)	tant que ($p2 \neq -1$) faire
	MOV (R2), (R3)	début $T3[libre].élément := T2[p2].élément$;
	ADD #1, R2	$T3[libre].suivant := libre+1$;
	MOV (R2), R2	libre := libre+1 ;
	ADD #1, R3	fin sinon
	ADD #2, R0	
	MOV R0, (R3)	
	ADD #1, R3	
	ADD #1, (R2)	
	JMP L6	
L4 :	SUB #1, (R1)	tant que ($p1 \neq -1$) faire
	MOV (R1), (R3)	début $T3[libre].élément := T1[p1].élément$;
	ADD #1, R1	$T3[libre].suivant := libre+1$;
	MOV (R1), R1	libre := libre+1 ;
	ADD #1, R3	fin ;
	ADD #2, R0	

```

MOV R0, (R3)
ADD #1, R3
ADD #1, (R1)
JMP L0
L5 : SUB #1, (R1)
SUB #1, (R2)
SUB tête3, R0      si (libre = 1) alors tête3 := -1 sinon
JZ R0, L7          début
SUB #1, R3
MOV #-1, (R3)      T3[libre-1].suivant := -1 ;
MOV tête3, R3      tête3 := 1 ;
JMP Fin            fin ;
L7 : MOV #-1, R3
Fin : STOP          fin ;
tête1 : 31
        tête1+2
        65
        tête1+4
        121
        -1
tête2 : 7
        tête2+2
        17
        tête2+4
        53
        tête2+6
        329
        -1
tête3 :

```

On suppose que les listes fournies en entrée apparaissent à la fin du programme respectivement aux adresses tête1 et tête2. A la fin de l'exécution de ce programme, tête3 contient le début de la liste résultat de la fusion des deux listes données en entrée.

- 5) La complexité de ce programme est identique à celle de l'algorithme de 1) à une constante multiplicative près. Elle est donc $O(l1+l2)$.

Exercice 2.6

```
type liste : enregistrement clé : entier ;
    suiv : ↑liste ;
    fin ;  
  
procédure liste-miroir ;
entrée : une liste l : ↑liste ;
sortie : la liste miroir lmiroir : ↑liste ;  
  
var prec, courant, succ :↑liste ;
début
    si l ≠ nil alors
        début prec := nil ;
            courant := l ;
            tant que courant↑.suiv ≠ nil faire
                début succ := courant↑.suiv ;
                    courant↑. suiv := prec ;
                    prec := courant ;
                    courant := succ ;
                fin ;
                courant↑. suiv := prec ;
                lmiroir := courant ;
            fin ;
    fin ;
```

La complexité de la procédure est $O(n)$ où n est la longueur de la liste.