Chapitre 1

Structures de données élémentaires

1.1 Introduction

En informatique, il existe plusieurs manières de représenter la notion mathématique d'ensemble. Il n'existe pas une représentation qui soit « meilleure » que les autres dans l'absolu : pour un problème donné, la meilleure représentation sera celle qui permettra de concevoir le « meilleur » algorithme, c'est-à-dire celui le plus esthétique et de moindre complexité. On parlera parfois d'*ensembles dynamiques* car nos ensembles seront rarement figés.

Chaque élément de ces ensembles pourra comporter plusieurs *champs* qui peuvent être examinés dès lors que l'on possède un *pointeur* —ou une *référence* si on préfère utiliser une terminologie plus proche de *Java* que de *C*— sur cet élément. Certains ensembles dynamiques supposent que l'un des champs de l'objet contient une **clé** servant d'identifiant.

Ces ensembles supportent potentiellement tout une série d'opérations :
- RECHERCHE(S, k) : étant donné un ensemble S et une clé k, le résultat de cette requête est un pointeur sur un élément de S de clé k, s'il en existe un, et la valeur NIL sinon —NIL étant un pointeur ou une référence sur « rien ».
- INSERTION(S, x) : ajoute à l'ensemble S l'élément pointé par x.
- SUPPRESSION(S, x) : supprime de l'ensemble S son élément pointé par x (si l'on souhaite supprimer un élément dont on ne connaît que la clé k, il suffit de récupérer un pointeur sur cet élément via un appel à RECHERCHE(S, k)).

Si l'ensemble des clés, ou l'ensemble lui-même, est totalement ordonné, d'autres opérations sont possibles :
- MINIMUM(S) : renvoie l'élément de S de clé minimale.
- MAXIMUM(S) : renvoie l'élément de S de clé maximale.
- SUCCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S pointé par x, et NIL dans le cas contraire.
- PRÉDÉCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S pointé par x, et NIL dans le cas contraire.

1.2 Piles et files

1.2.1 Piles

Définition 1 (Pile). *Une pile est une structure de données mettant en œuvre le principe « dernier entré, premier sorti » (*LIFO : Last-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération SUPPRESSION est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui le plus récemment inséré. L'opération INSERTION dans une pile est communément appelée EMPILER, et l'opération SUPPRESSION, DÉPILER. La figure 1.1 montre les conséquences des opérations EMPILER et DÉPILER sur une pile.

Il est facile d'implémenter une pile au moyen d'un tableau, comme le montre la figure 1.2. La seule difficulté dans cette implémentation est la gestion des débordements de pile qui interviennent quand on tente d'effecteur l'opération DÉPILER sur une pile vide et l'opération EMPILER sur un tableau codant la pile qui est déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les piles au moyen d'une structure de données dont

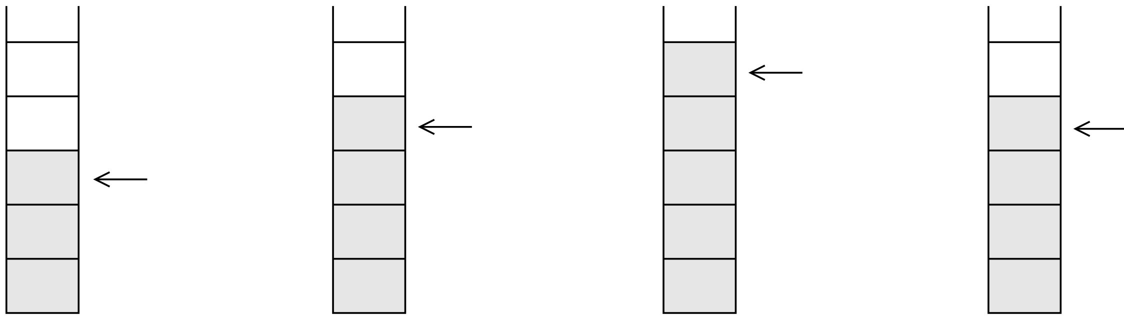

FIG. 1.1 – Exemple de pile : a) initialement la pile contient les valeurs 3, 5 et 2 ; b) état de la pile après l'opération EMPILER(6) ; c) état de la pile après l'opération EMPILER(1) ; d) état de la pile après l'opération DÉPILER, qui a renvoyé la valeur 1.

la taille n'est pas fixée *a priori* (comme une liste chaînée). Les algorithmes réalisant les fonctions EMPILER et DÉPILER, ainsi que la nécessaire fonction auxiliaire PILE-VIDE, sont présentés figure 1.3.

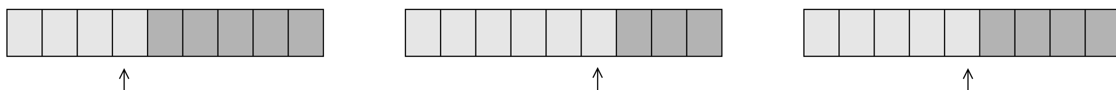

FIG. 1.2 – Implémentation d'une pile par un tableau : a) état initial de la pile ; b) nouvel état après les actions EMPILER(7) et EMPILER(3) ; c) nouvel état après l'opération DÉPILER qui a renvoyé la valeur 3.

1.2.2 Files

Définition 2 (File). *Une file est une structure de données mettant en œuvre le principe « premier entré, premier sorti » (*FIFO : First-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération SUPPRESSION est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui qui est resté le plus longtemps dans la file. Une file se comporte exactement comme une file d'attente de la vie courante. La figure 1.4 montre les conséquences des opérations INSERTION et SUPPRESSION sur une file.

On peut implémenter les files au moyen de tableaux. La figure 1.5 illustre l'implémentation de files à $n-1$ éléments au moyen d'un tableau à n éléments et de deux attributs :

- *tête*(F) qui indexe (ou pointe) vers la tête de la file ;
- *queue*(F) qui indexe le prochain emplacement où sera inséré un élément nouveau.

Les éléments de la file se trouvent donc aux emplacements *tête*(F), *tête*(F)+1, ..., *queue*(F)-1 (modulo n). Quand *tête*(F) = *queue*(F), la liste est vide. Les algorithmes réalisant les fonctions INSERTION et SUPPRESSION, ainsi que la nécessaire fonction auxiliaire FILE-VIDE, sont présentés figure 1.6. La seule difficulté dans cette implémentation est la gestion des débordements de file qui interviennent quand on tente d'effectuer l'opération SUPPRESSION sur une pile vide et l'opération INSERTION sur un tableau codant la file qui est déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les files au moyen d'une structure de donnée dont la taille n'est pas fixée *a priori* (comme une liste doublement chaînée).

PILE-VIDE(P)

si *sommet*(P)=0 **alors renvoyer** VRAI
sinon renvoyer FAUX

EMPILER(P, x)

si *sommet*(P) = *longueur*(P) **alors erreur** « débordement positif »
sinon *sommet*(P) ← *sommet*(P)+1
$P[sommet(P)] \leftarrow x$

DÉPILER(P)

si PILE-VIDE(P) **alors erreur** « débordement négatif »
sinon *sommet*(P) ← *sommet*(P)−1
renvoyer $P[sommet(P)+1]$

FIG. 1.3 – Algorithmes de manipulation des piles implémentées par des tableaux.

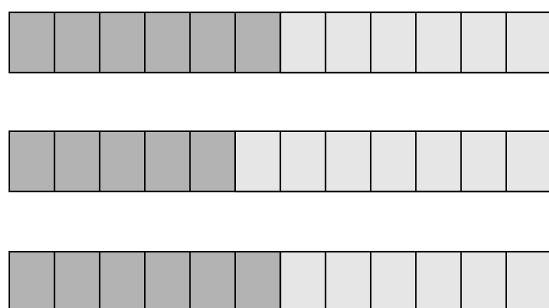

FIG. 1.4 – Exemple de file : a) initialement la file contient les valeurs 7, 4, 8, 9, 6 et 1 (de la plus anciennement à la plus récemment insérée) ; b) état de la file après l'opération INSERTION(3) ; c) état de la file après l'opération SUPPRESSION qui a renvoyé la valeur 7.

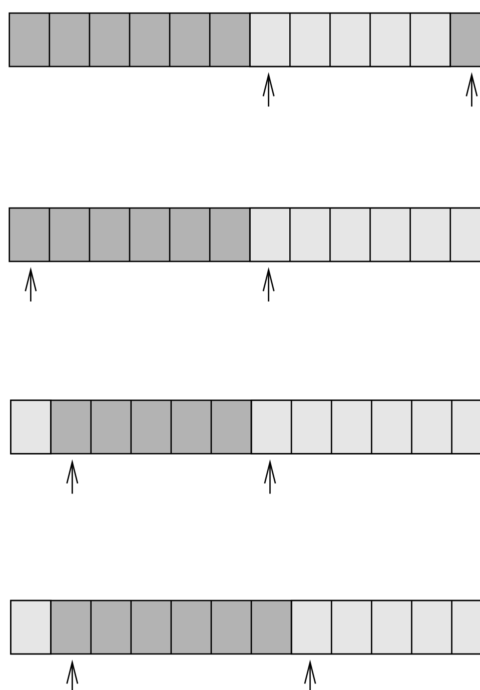

FIG. 1.5 – Implémentation d'une file par un tableau : a) état initial de la file ; b) nouvel état après l'action INSERTION(7) ; d) nouvel état après l'action INSERTION(5) ; d) nouvel état après l'opération SUPPRESSION qui a renvoyé la valeur 1.

FILE-VIDE(F)
  si *tête*(F)=*queue*(F) **alors renvoyer** VRAI
  sinon renvoyer FAUX

INSERTION(F, x)
  si *queue*(F) + 1 (modulo n) = *tête*(F) **alors erreur** « débordement positif »
  sinon $F[queue(F)] \leftarrow x$
  $queue(F) \leftarrow queue(F)+1$

SUPPRESSION(F)
  si FILE-VIDE(F) **alors erreur** « débordement négatif »
  sinon $tête(F) \leftarrow tête(F)+1$
  renvoyer $F[tête(F)-1]$

FIG. 1.6 – Algorithmes de manipulation des files implémentées par des tableaux.

1.3 Listes chaînées

1.3.1 Définitions

Définition 3 (Liste chaînée). *Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement, l'ordre linéaire étant déterminé par des pointeurs sur les éléments.*

Chaque élément de la liste, outre le champ *clé*, contient un champ *successeur* qui est pointeur sur l'élément suivant dans la liste chaînée. Si le champ *successeur* d'un élément vaut NIL, cet élément n'a pas de successeur et est donc le dernier élément ou la **queue** de la liste. Le premier élément de la liste est appelé la **tête** de la liste. Une liste L est manipulée via un pointeur vers son premier élément, que l'on notera TÊTE(L). Si TÊTE(L) vaut NIL, la liste est vide.

La figure 1.7 présente un exemple de liste chaînée et montre les conséquences des opérations INSERTION et SUPPRESSION sur une telle structure de données.

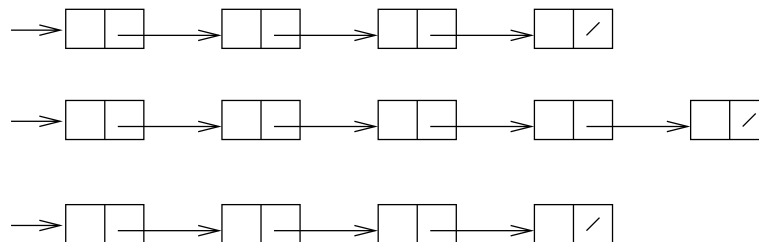

FIG. 1.7 – Exemple de liste chaînée : a) initialement la liste chaînée contient les valeurs 9, 6, 4 et 1 ; b) état de la liste chaînée après l'opération INSERTION(5) ; c) état de la liste chaînée après l'opération SUPPRESSION(4).

Une liste chaînée peut prendre plusieurs formes :

– **Liste doublement chaînée** : en plus du champ *successeur*, chaque élément contient un champ *prédécesseur* qui est un pointeur sur l'élément précédant dans la liste. Si le champ *prédécesseur* d'un élément vaut NIL, cet élément n'a pas de prédécesseur et est donc le premir élément ou la **tête** de la liste. Une liste qui n'est pas doublement chaînée est dite **simplement chaînée**.
  La figure 1.8 présente un exemple de liste doublement chaînée et montre les conséquences des opérations INSERTION et SUPPRESSION sur une telle structure de données.
– **Triée** ou **non triée** : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire des clés de ces éléments.
– **Circulaire** : si le champ *précécesseur* de la tête de la liste pointe sur la queue, et si le champ *successeur* de la queue pointe sur la tête. La liste est alors vue comme un anneau.

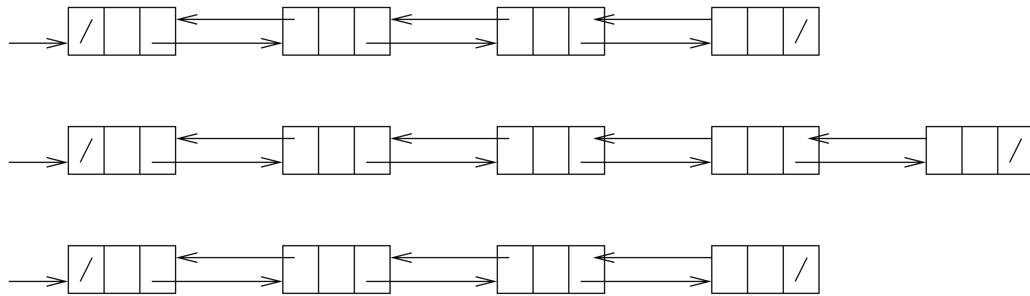

FIG. 1.8 – Exemple de liste doublement chaînée : a) initialement la liste contient les valeurs 9, 6, 4 et 1 ; b) état de la liste après l'opération INSERTION(5) ; c) état de la liste après l'opération SUPPRESSION(4).

1.3.2 Algorithmes de manipulation des listes chaînées

Recherche

L'algorithme RECHERCHE-LISTE(L, k) trouve le premier élément de clé k dans la liste L par une simple recherche linéaire, et retourne un pointeur sur cet élément. Si la liste ne contient aucun objet de clé k, l'algorithme renvoie NIL.

RECHERCHE-LISTE(L, k)
  $x \leftarrow$ TÊTE(L)
  tant que $x \neq$ NIL et *clé*(x) $\neq k$ **faire**
    $x \leftarrow$ *successeur*(x)
  renvoyer x

Cet algorithme manipule aussi bien des listes simplement que doublement que simplement chaînées.

Insertion

Étant donné un élément x et une liste L, l'algorithme INSERTION-LISTE insère x en tête de L.

INSERTION-LISTE(L, x)
  successeur(x) $\leftarrow$ TÊTE(L)
  si TÊTE(L) $\neq$ NIL **alors** *prédecesseur*(TÊTE(L)) $\leftarrow x$
  TÊTE(L) $\leftarrow x$
  prédecesseur(x) $\leftarrow$ NIL

Cet algorithme est écrit pour les listes doublement chaînées. Il suffit d'ignorer les deux instructions concernant le champ *prédécesseur* pour obtenir l'algorithme équivalent pour les listes simplement chaînées.

Suppression

L'algorithme SUPPRESSION-LISTE élimine un élément x d'une liste chaînée L. Cet algorithme a besoin d'un pointeur sur l'élément x à supprimer. Si on ne possède que la clé de cet élément, il faut préalablement utiliser l'algorithme RECHERCHE-LISTE pour obtenir le pointeur nécessaire.

SUPPRESSION-LISTE(L, x)
  si *prédécesseur*(x) $\neq$ NIL
    alors *successeur*(*prédécesseur*(x)) $\leftarrow$ *successeur*(x)
    sinon TÊTE(L) $\leftarrow$ *successeur*(x)
  si *successeur*(x) $\neq$ NIL
    alors *prédécesseur*(*successeur*(x)) $\leftarrow$ *prédécesseur*(x)

Cet algorithme est écrit pour les listes doublement chaînées. L'algorithme équivalent pour les listes simplement chaînées est plus compliqué puisqu'avec les listes simplement chaînées nous n'avons pas de moyen simple de récupérer un pointeur sur l'élément qui précède celui à supprimer...

SUPPRESSION-LISTE(L, x)
si x = TÊTE(L)
alors TÊTE(L) $\leftarrow$ *successeur*(x)
sinon $y \leftarrow$ TÊTE(L)
tant que *successeur*(y) $\neq x$ **faire** $y \leftarrow$ *successeur*(y)
successeur(y) $\leftarrow$ *successeur*(x)

1.3.3 Comparaison entre tableaux et listes chaînées

Aucune structure de données n'est parfaite, chacune a ses avantages et ses inconvénients. La figure 1.9 présente un comparatif des listes simplement chaînées, doublement chaînées et des tableaux, triés ou non, sur des opérations élémentaires. Les complexités indiquées sont celles du *pire cas*. Suivant les opérations que nous aurons à effectuer, et suivant leurs fréquences relatives, nous choisirons l'une ou l'autre de ces structures de données.

	liste chaînée simple non triée	liste chaînée simple triée	liste chaînée double non triée	liste chaînée double triée	tableau non trié	tableau trié
RECHERCHE(L, k)	$\Theta(n)^a$	$\Theta(n)^a$	$\Theta(n)^a$	$\Theta(n)^a$	$\Theta(1)^b$	$\Theta(1)^b$
INSERTION(L, x)	$\Theta(1)$	$\Theta(n)^e$	$\Theta(1)$	$\Theta(n)^e$	$\Theta(n)^c$ ou erreur^f	$\Theta(n)^d$ $\Theta(n)^c$ ou erreur^f
SUPPRESSION(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)^g$	$\Theta(n)^g$
SUCCESSEUR(L, x) ^{h}	$\Theta(n)^i$	$\Theta(1)$	$\Theta(n)^i$	$\Theta(1)$	$\Theta(n)^i$	$\Theta(1)$
PRÉDÉCESSEUR(L, x) ^{h}	$\Theta(n)^i$	$\Theta(n)^j$	$\Theta(n)^i$	$\Theta(1)$	$\Theta(n)^i$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)^i$	$\Theta(1)$	$\Theta(n)^i$	$\Theta(1)$	$\Theta(n)^i$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)^i$	$\Theta(n)^k$	$\Theta(n)^i$	$\Theta(n)^k$	$\Theta(n)^i$	$\Theta(1)$

[a] Dans le pire cas il faut parcourir tous les éléments pour se rendre compte que la clef n'était pas dans l'ensemble.
[b] La clé étant l'indice de l'élément dans le tableau.
[c] Dans le pire cas, il faut allouer un nouveau tableau et recopier tous les éléments de l'ancien tableau dans le nouveau.
[d] Dans le pire cas, l'insertion a lieu dans la première cas du tableau, et il faut décaler tous les éléments déjà présents.
[e] Au pire, l'insertion a lieu en fin de liste.
[f] Au cas où l'on veut effectuer une insertion dans un tableau déjà plein et qu'il n'est pas possible d'effectuer une allocation dynamique de tableau, comme en FORTRAN 77 ou en PASCAL.
[g] Dans le pire cas on supprime le premier élément du tableau et il faut décaler tous les autres éléments.
[h] Au sens de l'ordre sur la valeur des clés.
[i] Complexité de la recherche du maximum (ou du minimum) dans un ensemble à n éléments...
[j] Complexité de la recherche du maximum dans un ensemble à n éléments... car il faut entreprendre la recherche du prédécesseur depuis le début de la liste.
[k] Il faut parcourir la liste en entier pour trouver son dernier élément.

FIG. 1.9 – Efficacités respectives des listes chaînées et des tableaux.