Algorithmique III. L2 Informatique I41.

TD 10. Listes, piles, files et évaluation à l'aide d'une pile [1]

EXERCICE 1. On considère une liste L dont la structure de données associée `L` est une liste chaînée, une liste est donc une *référence*, elle n'est que la clé pour accéder à l'objet référencé noté ici `L»`. Les expressions `L»val` et `L»suiv` désignent respectivement la valeur contenue dans la cellule en tête de la liste L et la sous-liste suivante. On note `[]` la liste vide. On ne se préoccupe ni d'allocation mémoire ni de libération mémoire, contrairement au langage C.

(1) Écrivez les algorithmes suivants :

- `InsQueue(@L,x)` qui rajoute une cellule contenant `x` en queue de liste.
- `SupQueue(@L)` qui renvoie élimine la cellule en queue de liste et renvoie la valeur qu'elle contenait.
- `InsTete(@L,x)` qui rajoute une cellule contenant `x` en début de liste.
- `SupTete(@L)` qui élimine la cellule en tête de liste et renvoie la valeur qu'elle contenait.

(2) Implantez ces algorithmes sous forme de fonctions en langage C à l'aide des structures prédéfinies suivantes (le type `tval` n'est pas spécifié) :

....

```
typedef struct tcell{
   tval val;
   struct tcell *suiv;
} tcell;

typedef tcell *tliste;
```

....

Solution. On rappelle que les piles et les files ne sont rien d'autre que des listes. Ces qualificatifs ne dépendent pas de la structure en elle-même (ici une liste chaînée), mais de la manière dont les opérations d'insertion et de suppression sont réalisées. À la même extrémité, i.e. à l'aide du couple `InsTete`/`SupTete` ou du couple `InsQueue`/`SupQueue` pour une pile et aux extrémités opposées, i.e. à l'aide du couple `InsTete`/`SupQueue` ou `InsQueue`/`SupTete` pour une file.

(1) Le passage de la liste, qui est une *référence* par hypothèse, doit se faire ici *par adresse*. En effet, on suppose que la liste `L` peut être vide, auquel cas la référence est modifiée. Notons que si l'on faisait l'hypothèse que l'algorithme n'est jamais appelé avec une liste vide (imposant l'insertion du premier élément en amont, ce qui serait peu satisfaisant), le passage *par valeur* suffirait puisque seule la référence de la dernière cellule serait modifiée.

Le principe de l'algorithme `InsQueue` est simple, on crée une liste atomique `A` réduite à une seule cellule contenant `x` et la référence suivante vers la liste vide. Si `L` est vide, on la remplace par `A`, sinon il faut la parcourir et s'arrêter sur sa dernière cellule pour y accrocher `A`. Le passage de la liste se faisant par adresse, il faut se déplacer avec une autre variable que `L` (ici `I`). On sait que l'on a atteint la fin de la liste `L` si la référence *suivante* de la cellule courante `I` est la liste vide.

L'algorithme `SupQueue` suppose que la liste en entrée n'est pas vide. Le principe est le même que pour l'algorithme `InsQueue`, il faut parcourir toute la liste mais il faut cette fois s'arrêter sur l'*avant-dernière* cellule, dont la référence `suiv` sera la liste vide après la suppression de la dernière cellule. Le problème est plus simple pour le duo `InsTete` et `SupTete` puisqu'il n'y a pas à parcourir la liste. L'algorithme `SupTete` suppose que la liste en entrée a été testée en amont et n'est pas vide.

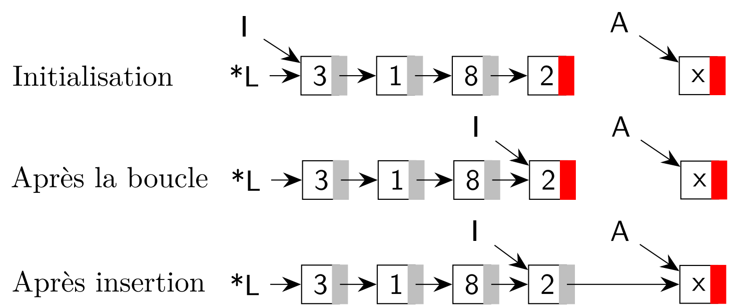

FIGURE 1. Insertion d'une cellule en queue de liste.

1. Version du 28 avril 2023, 14 : 23

```
ALGORITHME InsQueue(@L,x)
DONNEES
 · L: liste
 · x: valeur
VARIABLES
 · I,A: listes
DEBUT
 · A» ← {x, []}
 · A»suiv ← []
 · I ← L
 · SI (I = []) ALORS
 ·   · L ← A
 · SINON
 ·   · TQ (I»suiv != []) FAIRE
 ·   ·   · I ← I»suiv
 ·   · FTQ
 ·   · I»suiv ← A
 · FSI
 FIN
```

```
ALGORITHME SupQueue(@L):valeur
DONNEES
 · L: liste
VARIABLES
 · I: liste
 · x: valeur
DEBUT
 · I ← L
 · SI (L»suiv = []) ALORS
 ·   · x ← L»val
 ·   · L ← []
 · SINON
 ·   · TQ ((I»suiv)»suiv != []) FAIRE
 ·   ·   · I ← I»suiv
 ·   · FTQ
 ·   · x ← (I»suiv)»val
 ·   · I»suiv ← []
 · FSI
 · RENVOYER x
FIN
```

Algo. 1. Insertion et suppression en queue de liste

(2) La traduction en *C* demande un petit effort. D'une part, parce que ce langage ne connaît que le passage des paramètres par valeur, (contrairement à notre pseudo-langage où le passage par adresse se fait en précédant une variable par `@`), problème contourné en *C* en passant la valeur de l'adresse `&x` d'une variable `x` nécessitant son déréférencement `*x` dans le corps de la fonction. D'autre part, il faut allouer la mémoire pour chaque cellule et la restituer quand on défile ou on dépile.

La figure 1 explicite le déroulement de `InsQueue` pour une liste non-vide. Une cellule est matérialisée par une boite contenant la valeur et la référence suivante par une bande grise et une flèche, ou une simple bande rouge si elle est vide. La 1ère ligne montre l'état des listes après l'initialisation de `A` et `I`, la deuxième après la boucle `while` et la dernière après l'insertion de `A`.

```
ALGORITHME InsTete(@L,x)
DONNEES
 · L: liste
 · x: valeur
VARIABLES
 · I,A: listes
DEBUT
 · A» ← {x, []}
 · L ← A
FIN
```

```
ALGORITHME SupTete(@L):valeur
DONNEES
 · L: liste
VARIABLES
 · x: valeur
DEBUT
 · x ← L»val
 · L ← L»suiv
 · RENVOYER x
FIN
```

Algo. 2. Insertion et suppression en tête de liste.

```c
void InsQueue(tliste *L, tval x){
   tliste A = malloc(sizeof(tcell));
   tliste I = *L;
   *A = {x, NULL};
   if (I == NULL)
      *L = A;
   else{
      while (I->suiv != NULL)
         I = I->suiv;
      I->suiv = A;
   }
}
```

```c
tval SupQueue(tliste *L){
   tval x = 0;
   tliste I = *L;
   if (I->suiv == NULL){
      x = I->val;
      free(*L);
      *L = NULL;
   }
   else{
      while ((I->suiv)->suiv
               != NULL)
         I = I->suiv;
      x = (I->suiv)->val;
      free(I->suiv);
      I->suiv = NULL;
   }
   return x;
}
```

Algo. 3. Insertion et suppression en queue de liste (C).

La figure 2 montre l'évolution de l'algorithme `SupQueue` dans le cas où la liste contient déjà des valeurs (entières). La première ligne montre l'état des listes après l'initialisation des variables `A` et `I`, la deuxième après la boucle `while` et la dernière après l'insertion de la liste `A`.

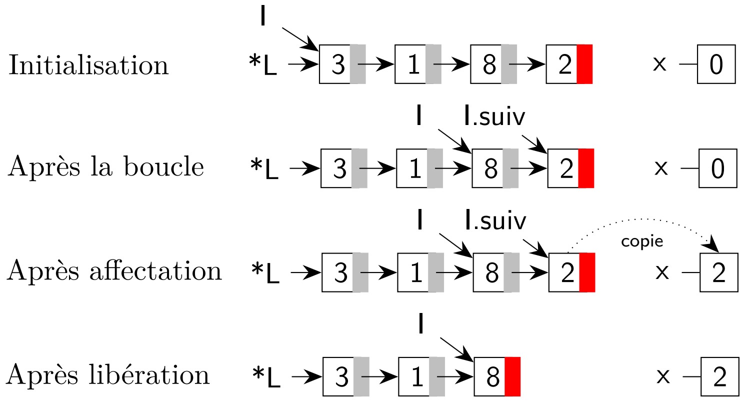

FIGURE 2. Suppression d'une cellule en queue de liste.

Le fonctionnement des deux algorithmes `InsTete` et `SupTete` est plus simple et l'illustration est laissée au lecteur. Les fonctions C correspondantes sont :

```
void InsTete(tliste *L, tval x){
   tliste A = malloc(sizeof(tcell));
   A->val = x;
   A->suiv = *L;
   *L = A;
}
```

```
tval SupTete(tliste *L){
  tval x = (*L)->val
  tliste I = *L;
  *L = I->suiv;
  free(I);
  return x;
}
```

ALGO. 4. Insertion et suppression en tête de liste (C).

EXERCICE 2. Un mot sur l'alphabet $\Sigma := \{$`{`, `(`, `[`, `)`, `]`, `}`$\}$ est bien parenthésé s'il est défini inductivement par l'une des règles de construction suivantes :

(a) mot := ε (le mot vide).

(b) mot := $\langle$mot$\rangle$ où $(\langle, \rangle) \in \{$(`{`, `}`), (`(`, `)`), (`[`, `]`)$\}$.

(c) mot := mot.mot (concaténation).

Par exemple, le mot `{(())}[()]` est bien parenthésé alors que les mots `())()` et `[{]}}` ne le sont pas.

(1) Écrivez un algorithme `EstBP(mot)` qui utilise une pile pour décider si un `mot` de Σ^* est bien parenthésé ou non.

(2) Quelle est sa complexité ?

(3) S'il n'y avait qu'un seul type de parenthèses, pourrait-on procéder plus simplement ?

Solution. Une pile initialement vide permet de vérifier que les parenthèses d'un mot sont correctement appariées de la manière suivante : on parcourt les symboles x_i du mot $x_1x_2 \dots x_n$ et si

(a) $x_i \in \{$`{`, `(`, `[`$\}$, alors on empile x_i.

(b) $x_i \in \{$`}`, `)`, `]`$\}$, alors le sommet de la pile doit contenir la parenthèse ouvrante correspondante et dans ce cas on dépile, sinon l'expression n'était pas bien parenthésée.

À chaque fois que toutes les parenthèses ouvrantes ont été correctement fermées, la pile est vide et à la fin de l'analyse, la pile doit être vide. Voir le traitement du mot `{(())}[()]` en table 1.

3				(								
2			(	(	(				(			
1		{	{	{	{	{		[	[	[		
x_i		{	(	(	)	)	}	[	(	)	]	

TABLE 1. Empilement et dépilement des parenthèses ouvrantes.

Dans l'algorithme ci-dessous, les algorithmes `InsTete` et `SupTete` ont été rebaptisés `Empiler` et `Depiler` et pour ne pas alourdir les écritures, la parenthèse opposée à une parenthèse p est noté $-p$.

(1) La complexité est évidemment en $\Theta(n)$ où n est la longueur du mot à analyser.

```
ALGORITHME EstBP(mot):booléen
DONNEES
 · mot: chaîne
VARIABLES
 · i: entier
 · PILE: pile
 · R: booléen
DEBUT
 · PILE ← []
 · R ← VRAI
 · i ← 1
 · TQ ((i <= #mot) ET R) FAIRE
 ·   · SI (mot[i] DANS {"{", "(", "["}) ALORS
 ·   ·   · Empiler(PILE, mot[i])
 ·   · SINON
 ·   ·   · SI ((PILE != []) ET (PILE»val = -mot[i])) ALORS
 ·   ·   ·   · Depiler(PILE)
 ·   ·   · SINON
 ·   ·   ·   · R ← FAUX
 ·   ·   · FSI
 ·   · FSI
 ·   · i ← i + 1
 · FTQ
 · RENVOYER R
FIN
```

Algo. 5. Vérification de correction de parenthésage.

(2) S'il n'y avait qu'un seul type de parenthèse, il n'y aurait pas à tester si la parenthèse fermante est l'opposée de la parenthèse ouvrante au sommet de la pile, mais dans ce cas un simple compteur `hauteur` simulant la hauteur de la pile suffirait à décider si l'expression est bien parenthésée ou non. On remplacerait l'empilement et le dépilement par l'incrémentation et la décrémentation de la hauteur respectivement et la condition `(PILE != [])` de boucle serait remplacée par `(hauteur != 0)`.

EXERCICE 3. Écrivez un algorithme `Inverse(@pile)` qui inverse l'ordre des éléments de la pile passée en paramètre et *in situ*. Écrivez cet algorithme en langage *C*.

Solution. On suppose que la pile a été obtenue en opérant sur la tête de la liste chaînée. Dans ce cas, rien de bien compliqué, il faut simplement décrocher une à une les cellules en tête de la pile pour les accrocher en tête d'une liste initialement vide, renversant ainsi l'ordre des valeurs. On pourrait utiliser les algorithmes `SupTete` et `InsTete`, mais ils opèrent sur les valeurs dans la cellule de tête et pas sur la cellule elle-même. La conversion en langage *C* est immédiate et ne pose aucune difficulté (avec les mêmes remarques sur le passage de la liste "par adresse").

```
ALGORITHME Inverser(@pile)        void Inverser(tliste *pile){
DONNEES                          I>   tliste A = NULL;
 · pile: liste                   I>   tliste I = *pile;
VARIABLES                        I>   *pile = NULL;
 · A,I: liste                         while (I != NULL){
DEBUT                            a>      A = I;
 · I ← pile                      b>      I = I->suiv;
 · pile ← []                     c>      A->suiv = *pile;
 · TQ (I != []) FAIRE            d>      *pile = A;
 ·   · A ← I                          }
 ·   · I ← I»suiv                  }
 ·   · A»suiv ← pile
 ·   · pile ← A
 · FTQ
FIN
```

Algo. 6. Inversion d'une pile.

Dans le cas où la `pile` est vide, elle le reste puisque `I` est vide et on n'entre pas dans la boucle. La figure 3 montre l'évolution des variables entre la fin de l'initialisation et un premier passage dans la boucle.

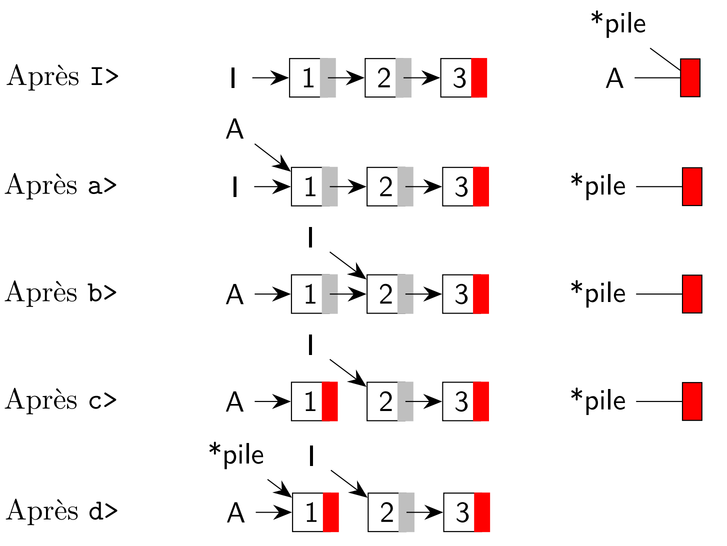

Figure 3. Renversement d'une pile.

EXERCICE 4. Dans la suite `L` désigne une liste chaînée.

(1) Écrivez un algorithme `Inserer(@L,pos,A)` qui insère la liste atomique `A` *après* la cellule référencée par `pos` de la liste `L`. On suppose que `A` peut désigner n'importe quelle cellule de la liste `L`. Écrivez cet algorithme en langage *C*.

(2) Écrivez une fonction `tliste Fusionner(tliste *A, tliste *B)` qui fusionne deux listes triées `A` et `B` et renvoie la liste fusionnée triée à partir des cellules de `A` et `B` qui seront vides à l'issue du fusionnement.

Solution. **(1)** Il faut être attentif à traiter le cas où la liste `L` est vide. Dans ce cas, la liste `pos` est vide également par hypothèse et la liste `L` doit simplement être initialisée à la liste atomique `A` :

La figure 4 explicite le fonctionnement de l'algorithme en langage *C* dans le cas où la liste n'est pas vide.

(2) Pour réaliser la fusion des deux listes triées `A` et `B`, on va décrocher la tête de celle qui contient la plus petite valeur (mémorisée dans la variable

```
ALGORITHME Inserer(@L,pos,A)        void Inserer(tliste *L,
DONNEES                                           tliste pos,
 · L,pos,A: listes                                tliste A){
DEBUT                                      if ((*L) == NULL)
 · SI (L = []) ALORS                          (*L) = A;
 ·   · L ← A                               else{
 · SINON                            1>        A->suiv = pos->suiv;
 ·   · A»suiv ← pos»suiv            2>        pos->suiv = A;
 ·   · pos»suiv ← A                        }
 · FSI                                   }
FIN
```

Algo. 7. Insertion d'une liste atomique en fin d'une liste.

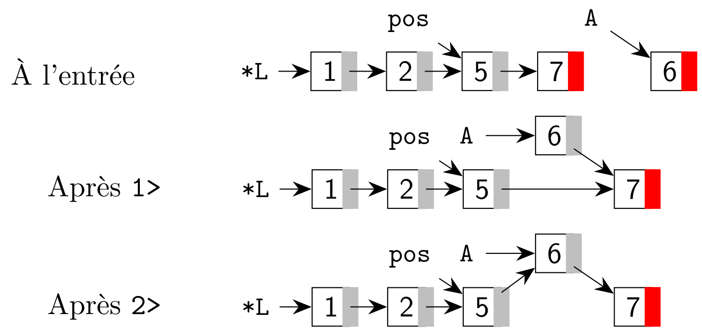

Figure 4. Insertion d'une liste atomique à une position donnée.

`aux` de la fonction *C* ci-dessous) pour l'insérer à la fin d'une liste `F` initialement vide et on recommence jusqu'à ce que l'une ou l'autre des deux listes `A` ou `B` soit vide, auquel cas il suffit de concaténer l'autre à la queue de la liste `F`.

EXERCICE 5. Écrivez l'algorithme du tri insertion en langage *C* qui trie les valeurs d'une liste *doublement* chaînée dans l'ordre croissant. Indications : il suffit de rajouter le champ `struct tcell *prec;` dans la structure d'une cellule pour créer un chaînage double.

```
tliste Fusionner(tliste *A, tliste *B){
   tliste F = NULL;
   tliste ou = F;
   tliste aux = NULL;
   while ((*A != NULL) && (*B != NULL)){
      if ((*A)->val <= (*B)->val){
         aux = *A;
         *A = (*A)->suiv;
      }
      else{
         aux = *B;
         *B = (*B)->suiv;
      }
      aux->suiv = NULL;
      Inserer(&F,ou,aux);
      ou = aux;
   }
   ou->suiv = ((*A) == NULL) ? *B : *A;
   *A = *B = NULL;
   return F;
}
```

Algo. 8. Fusionnement de deux listes (C).

Solution. Le principe est calqué sur l'algorithme développé sur une structure de liste statique. On se déplace le long de la liste en "insérant" la valeur courante de la droite vers la gauche (d'où la nécessité du chaînage double pour reculer dans la liste). Mais plutôt que d'échanger deux cellules contiguës de la liste, on échange simplement leurs valeurs. Notons que le passage "par adresse" n'est pas nécessaire en C puisque l'on échange les contenus des cellules et pas les cellules elles-mêmes.

EXERCICE 6. On dispose d'une mémoire M de taille n codée par un tableau `M` indexé de 1 à n de cellules à deux champs, un booléen `libre` indiquant si la cellule est libre et une valeur `val` contenant l'information à stocker. Initialement toutes les cellules sont libres.

```
void TriInsertion(tliste L){
   if (L != NULL){
      tliste I = L->suiv;
      while (I != NULL){
         tliste R = I;
         while ((R->prec != NULL) &&
                (R->val < (R->prec)->val)){
            uint aux = (R->prec)->val;
            (R->prec)->val = R->val;
            R->val = aux;
            R = R->prec;
         }
         I = I->suiv;
      }
   }
}
```

Algo. 9. Tri insertion sur liste chaînée (C).

(1) Écrivez un algorithme `Reserver(i,k)` (resp. `Liberer(i,k)`) qui réserve (resp. libère) les k cellules mémoire à partir de la position `i` en instanciant leurs champs `libre` à *faux* (resp. *vrai*).

(2) Écrivez un algorithme `ChercherLibre(k)` qui renvoie le plus petit indice i tel que les cellules $M[i+r]$ sont libres pour tout $r \in [\![0, k-1]\!]$. L'algorithme renvoie 0 si aucun bloc de taille k n'est libre.

(2) Faites une preuve d'arrêt et justifiez l'algorithme.

(3) Écrivez un algorithme `Allouer(k,@i)` qui cherche l'adresse i du premier bloc de k cellules libres de la mémoire et les réserve puis renvoie *vrai* s'il a pu les trouver et *faux* sinon.

(4) Calculez la/les complexité(s) de l'algorithme d'allocation.

Solution. **(1)** L'écriture des algorithmes de réservation et de libération de la mémoire ne posent aucune difficulté. On suppose ici que la condition $k+i-1 \leqslant n$ assurant que l'on ne réserve ou ne libère pas des cellules hors de la mémoire a été testée en amont :

(2) L'écriture de l'algorithme demande un peu de rigueur dans la gestion des indices. On utilise un compteur `c` pour mesurer la taille d'un bloc

```
ALGORITHME Reserver(i,k)
DONNEES
 · i,k: entiers
VARIABLES
 · r: entier
DEBUT
 · r ← 0
 · TQ (r < k) FAIRE
 ·   · M[i + r].libre ← FAUX
 ·   · r ← r + 1
 · FTQ
FIN
```

```
ALGORITHME Liberer(i,k)
DONNEES
 · i,k: entiers
VARIABLES
 · r: entier
DEBUT
 · r ← 0
 · TQ (r < k) FAIRE
 ·   · M[i + r].libre ← VRAI
 ·   · r ← r + 1
 · FTQ
FIN
```

Algo. 10. Réservation et libération de mémoire.

```
ALGORITHME ChercherLibre(k):entier
DONNEES
 · k: entier
VARIABLE
 · i, c: entiers
DEBUT
 · i ← 1
 · c ← 0   ·
 · TQ ((c < k) ET (i <= n - (k - c) + 1)) FAIRE
 ·   · SI M[i].libre ALORS
 ·   ·   · c ← c + 1
 ·   · SINON
 ·   ·   · c ← 0
 ·   · FSI
 ·   · i ← i + 1
 · FTQ
 · SI (c < k) ALORS
 ·   · RENVOYER 0
 · SINON
 ·   · RENVOYER i - k
 · FSI
FIN
```

Algo. 11. Recherche d'une zone libre.

libre. On parcourt la mémoire M tant que le compteur `c` n'a pas atteint la taille requise k en l'incrémentant si la cellule courrante est libre, sinon on le réinitialise à 0. La condition sur la variable `i` permet de s'assurer qu'il reste assez d'espace mémoire (i.e. `(k - c)` cellules libres) pour compléter le bloc. La mémoire est globale et est indexée de 1 à n.

(3) La variable variable c, initialement nulle, est éventuellement incrémentée (uniquement dans la boucle) mais la condition de boucle $c < k$ assure que $k - c \geqslant 0$ à tout moment. Ceci entraîne que la valeur maximale de l'expression $n-(k-c)+1$ est égale à $n+1$. Comme i, initialement nulle, est incrémentée à chaque passage dans la boucle, l'inégalité $i \leqslant n-(k-c)+1$ sera nécessairement invalidée et on sortira de la boucle.

La condition $c < k$ étant à gauche de la conjonction, la borne $n + 1$ n'est jamais atteinte par i, assurant que l'on ne débordre jamais de la mémoire. Montrons que s'il existe un bloc $M[p : p + k - 1]$ de k cellules libres ($k > 0$) dans la mémoire dont la première est à l'adresse p et qu'il n'existe aucun bloc libre de taille supérieure ou égale avant p, alors à la sortie de la boucle, $i - k = p$.

Montrons qu'à chaque passage dans la boucle, le bloc $M[i-c, i-(c+1)]$ est vide ($c = 0$) ou libre. C'est vrai avant d'entrer dans la boucle puisque le bloc est vide. Supposons que ce soit vrai à la i-ème entrée. Dans ce cas, soit la cellule i n'est pas libre et $c \leftarrow 0$ et dans ce cas le bloc suivant $M[i + 1, i]$ est vide, sinon c est également incrémentée et le bloc suivant $M[i + 1 - c, i + 1 - (c + 1)]$ est libre. Par hypothèse, la cellule p est libre et la précédente, si $p > 0$ ne l'est pas. Ainsi, quand la variable i atteint la valeur p, le compteur $c = 0$ et les $k - 1$ cellules suivantes étant libres par hypothèse, on incrémente i et c jusqu'à ce que $c = k$ puisque le bloc $M[p : p + k - 1]$ est libre. On sort de la boucle avec $k = c$ et $i = p + k$, donc $i - k = i - c = p$ et $M[i - c, i - (c + 1)] = M[p, p + k - 1]$.

(4) Il suffit de combiner la recherche d'un bloc libre et de réserver la mémoire correspondante.

```
ALGORITHME Allouer(k,@i):booléen
DONNEES
 · k: entier
VARIABLE
 · i: entier
DEBUT
 · i ← ChercherLibre(k)
 · SI (i > 0) ALORS
 ·    · Reserver(i,k)
 · FSI
 · RENVOYER (i > 0)
FIN
```

Algo. 12. Allocation mémoire.

(5) Dans le meilleur des cas, les k premières cellules de la mémoire sont libres, mais il aura fallu les parcourir une première fois pour le savoir et une seconde fois pour les réserver, donc $\check{T}(n) = \Theta(k)$. En première analyse, on pourrait penser que le pire des cas correspond à l'absence de bloc libre de taille k dans la mémoire, obligeant à la parcourir entièrement, mais c'est la situation où le bloc de k cellules libres est à la fin, nécessitant non seulement le parcours de toute la mémoire mais la réservation des k dernières cellules en sus. On a donc $\widehat{T}(n) = \Theta(n) + \Theta(k) = \Theta(n)$.

Notons que l'étude du cas moyen nécessiterait la mise en place d'un modèle probabiliste pour savoir quelle est la distribution et l'ordre dans lequel apparaissent les demandes d'allocation mémoire. Il faudrait également intégrer les restitutions de mémoire aboutissant souvent à de la *fragmentation* (il reste de la mémoire libre mais les blocs de cellules libres contiguës sont de petites tailles), remettant ainsi en question le mécanisme d'allocation tel qu'il a été proposé dans cet exercice, etc.

EXERCICE 7. Écrivez les expressions arithmétiques suivantes sous forme postfixe et calculez leurs valeurs à l'aide d'une pile et représentez l'évolution de cette pile lors de l'évaluation. Utilisez le symbole $\sim$ pour distinguer l'opérateur unaire moins de l'opérateur binaire $-$ dans les expressions postfixes.

(a) $(3-6)(2+1)$
(b) $4(1+2+3)$
(c) $(1+3)-5+(2+(-1))$
(d) $3+2((7+1)-(5+2))(-2)$.

Solution. Les jetons des expressions postfixes sont placés sous les piles.

3									
2						1			
1			6		2	2	3		
0		3	3	-3	-3	-3	-3	-9	
$\uparrow$	E	3	6	$-$	2	1	$+$	$*$	

Table 2. Pile d'évaluation de (a).

3									
2				2		3			
1			1	1	3	3	6		
0		4	4	4	4	4	4	24	
$\uparrow$	E	4	1	2	$+$	3	$+$	$*$	

Table 3. Pile d'évaluation de (b).

3												
2								1	-1			
1			3		5		2	2	2	1		
0		1	1	4	4	-1	-1	-1	-1	-1	0	
$\uparrow$	E	1	3	$+$	5	$-$	2	1	$\sim$	$+$	$+$	

Table 4. Pile d'évaluation de (c).

5																
4								2								
3					1		5	5	7							
2				7	7	8	8	8	8	1		2	-2			
1			2	2	2	2	2	2	2	2	2	2	2	-4		
0		3	3	3	3	3	3	3	3	3	3	3	3	3	-1	
$\uparrow$	E	3	2	7	1	$+$	5	2	$+$	$-$	$*$	2	$\sim$	$*$	$+$	

Table 5. Pile d'évaluation de (d).