Chapitre 7

Les piles et les files

Dans ce chapitre, nous n'allons pas introduire de nouveau mécanisme de programmation, mais appliquer ceux qui ont été étudiés aux chapitres précédents afin de développer deux structures de données importantes : les piles et les files. Nous allons montrer en particulier que ces structures peuvent être implémentées de différentes façons, tout en présentant la même interface au code qui les utilise.

7.1 Les structures de données

Dans le contexte particulier du langage C, nous avons introduit les données structurées au chapitre 6. En informatique, le concept de *structure de données* est plus général, et peut être vu comme une extension de la notion de *type* discutée pour la première fois au chapitre 2.

Le fait d'attribuer un type à une variable précise deux choses : la nature des données qui peuvent être placées dans cette variable (par exemple, en langage C, une variable de type `double` peut retenir la valeur approximée d'un nombre réel), et l'ensemble des opérations que l'on peut appliquer à ces données (par exemple, l'opérateur "`/`" appliqué à des valeurs de type `double` indique une division réelle).

De la même façon, la définition d'une *structure de données* spécifie la nature des données que les instances de cette structure retiennent, ainsi que la liste des opérations qu'il est possible de leur appliquer.

Une structure de données possède une *interface* et une *implémentation*, tout comme les fonctions étudiées au chapitre 4. L'interface est une documentation permettant d'exploiter la structure de données, en caractérisant rigoureusement les données qu'elle contient et en donnant le mode d'emploi des opérations qu'elle permet d'effectuer. L'implémentation précise quant à elle tous

les détails internes de la représentation des données au sein de la structure et des opérations qu'elle définit. Pour utiliser une structure de données, il est suffisant de connaître son interface.

7.2 Les piles

7.2.1 Définition

Une *pile (stack)* est une structure de données capable de retenir une collection de valeurs. Cette structure offre la possibilité d'ajouter un nouvel élément à la collection, de retirer un élément, et d'obtenir certaines informations sur le contenu courant de la collection. Une pile est caractérisée par le fait que les valeurs qu'elle contient ne sont accessibles que selon une politique *LIFO (Last In First Out)*. Cela signifie qu'on ne peut à chaque instant retirer de la pile que l'élément qui, parmi tous ceux qu'elle contient, a été ajouté en dernier lieu. En d'autres termes, une structure de pile permet d'accéder à ses éléments dans l'ordre inverse où ils y ont été placés.

Pour bien comprendre le principe de fonctionnement d'une pile, on peut considérer l'analogie entre une telle structure et une pile de crêpes : lorsqu'on cuisine des crêpes, on effectue deux opérations de base qui consistent à cuire des crêpes et à les manger. On peut respectivement comparer ces opérations à l'ajout et au retrait d'éléments à la structure.

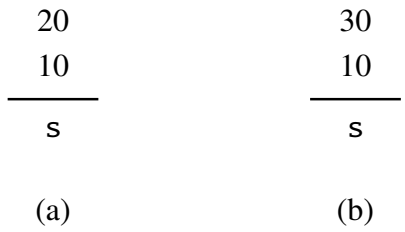
Si l'on ne dispose que d'une seule assiette pour y placer les crêpes, le seul endroit où l'on peut placer une crêpe que l'on vient de cuire est au-dessus des crêpes déjà présentes sur l'assiette, ou bien directement sur l'assiette s'il s'agit de la première crêpe. Pour manger une crêpe, la seule possibilité consiste à prendre celle qui se situe au-dessus de toutes les autres. Il s'agit d'une politique LIFO : on mange toujours la crêpe cuisinée en dernier lieu parmi celles présentes sur l'assiette.

7.2.2 Interface

Une structure de pile possède l'interface suivante. Nous ne faisons pas d'hypothèse sur la nature des valeurs que la pile retient, qui peuvent être de n'importe quel type. Les opérations que l'on peut appliquer à une pile s sont les suivantes.

— `push(`s`,` v`)` : *Empile* la valeur v, en d'autres termes, ajoute v à la collection de valeurs retenue par la pile s.

— `pop(`s`)` : *Dépile* une valeur, en d'autres termes, retire une valeur de la pile s et retourne cette valeur. Conformément à la politique d'accès LIFO, la valeur extraite est, parmi celles présentes sur la pile, celle qui a été empilée en dernier lieu. Cette opération signale une erreur si elle est appliquée à une pile vide.



(a) (b)

FIGURE 7.1 – Exemple de contenus de pile

— `top(`*s*`)` : Retourne la valeur située au *sommet* de la pile *s*, c'est-à-dire, celle empilée en dernier lieu parmi les valeurs présentes sur la pile, mais sans la dépiler. Cette opération ne modifie donc pas le contenu de la pile. Elle signale une erreur si elle est effectuée alors que la pile est vide.

— `size(`*s*`)` : Retourne le nombre de valeurs contenues dans la pile *s*.

— `is_empty(`*s*`)` : Retourne une valeur booléenne indiquant si la pile *s* est vide.

Pour illustrer le fonctionnement d'une pile, nous considérons la séquence suivante d'opérations, impliquant une pile `s` que nous supposons être initialement vide.

```
push(s, 10)
push(s, 20)
pop(s)
push(s, 30)
pop(s)
pop(s)
```

Les deux premières opérations empilent successivement les valeurs 10 et 20 ; après ces opérations, le contenu de la pile correspond à celui montré à la figure 7.1 (a). L'opération suivante dépile une valeur, et retourne donc celle empilée en dernier lieu, c'est-à-dire 20. Ensuite, on empile la valeur 30, ce qui mène au contenu de la pile de la figure 7.1 (b). Les deux dernières opérations dépilent les deux valeurs présentes sur la pile, dans l'ordre inverse de leur empilement, et retournent donc successivement 30 et 10.

7.2.3 Implémentation à l'aide d'un vecteur

Nous nous intéressons maintenant à la question d'implémenter une structure de pile, c'est-à-dire de développer un procédé pour représenter son contenu en mémoire, ainsi que d'écrire des fonctions pour chacune des 5 opérations qui ont été définies. Ce problème admet plusieurs

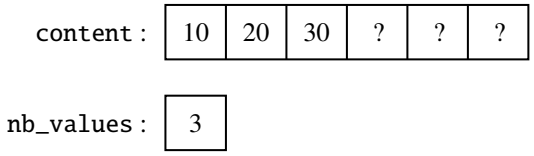


FIGURE 7.2 – Représentation d'une pile basée sur un vecteur

solutions, possédant chacune des avantages et des inconvénients. Nous en présentons une ici, et une alternative sera ensuite proposée à la section 7.2.5.

Une première façon de représenter une pile consiste à placer ses éléments successifs dans un vecteur. La taille de ce vecteur correspondra à la capacité maximale de la pile, c'est-à-dire au plus grand nombre d'éléments qu'elle peut contenir à chaque instant. Lorsque la pile atteint cette capacité maximale, il n'est plus possible d'effectuer une opération `push`. Cette opération échouera alors en signalant une erreur.

Représenter les éléments de la pile dans ceux d'un vecteur n'est pas suffisant ; il faut aussi retenir une donnée supplémentaire indiquant la position du sommet de la pile dans ce vecteur, ou de façon équivalente le nombre d'éléments contenus dans la pile. La figure 7.2 montre la représentation d'une structure décrivant une pile de capacité maximale égale à 6, contenant les valeurs 10, 20 et 30 empilées dans cet ordre. Le champ `content` correspond au tableau d'éléments, et le champ `nb_values` au nombre d'éléments de la pile.

Dans la structure de la figure 7.2, les valeurs appartenant à la pile sont directement placées dans les éléments du vecteur `content`. Le problème de cette approche est que le type de ces valeurs doit correspondre au type défini pour les éléments de ce vecteur. Si l'on souhaite disposer d'une structure de pile capable d'accepter des valeurs de n'importe quel type, une solution consiste à placer dans les éléments du vecteur des pointeurs vers les valeurs à retenir plutôt que ces valeurs. En effet, nous avons vu à la section 6.2.2 qu'il est possible de définir des pointeurs vers n'importe quelle donnée en employant le type `void *`. Ce type sera donc celui spécifié pour les éléments du vecteur `content`.

Un programme implémentant une structure de pile à l'aide d'un vecteur est donné aux figures 7.3 et 7.4. Comme d'habitude, son code source est réparti dans un fichier d'en-tête `stack.h` qui définit le type de la structure de données et le prototype des fonctions permettant de la manipuler, et un fichier `stack.c` qui contient l'implémentation de ces fonctions.

En addition aux 5 fonctions discutées à la section 7.2.2, ce programme définit des fonctions `stack_new` et `stack_free` qui permettent respectivement d'allouer une nouvelle pile (initialement vide) et de libérer une pile. La capacité de la pile est fixée par une constante `MAX_STACK_SIZE` définie dans le fichier d'en-tête `stack.h`.

stack.c:

```
#include <stdlib.h>
#include "stack.h"

/*  Retourne une nouvelle pile vide, ou NULL en cas d'erreur.  */

stack *stack_new()
{
  stack *s;

  s = malloc(sizeof(stack));
  if (!s)
    return NULL;

  s -> nb_values = 0;

  return s;
}

/*  Libère la pile <s>.  */

void stack_free(stack *s)
{
  free(s);
}

/*  Empile le pointeur <p> sur la pile <s>.  Retourne 0 en cas de succès,
    ou -1 si la pile est saturée.  */

int stack_push(stack *s, void *p)
{
  if (s -> nb_values >= MAX_STACK_SIZE)
    return -1;

  s -> content[s -> nb_values++] = p;
  return 0;
}

/*  Dépile un pointeur depuis la pile <s> et retourne ce pointeur,
    ou NULL si la pile est vide.  */

void *stack_pop(stack *s)
{
  if (!(s -> nb_values))
    return NULL;

  return s -> content[--(s -> nb_values)];
}

/*  Retourne le pointeur au sommet de la pile <s>, sans le dépiler,
    ou NULL si la pile est vide.  */

void *stack_top(stack *s)
{
  if (!(s -> nb_values))
    return NULL;

  return s -> content[s -> nb_values - 1];
}
```

FIGURE 7.3 – Implémentation d'une pile basée sur un vecteur (1/2)

```
  /*  Retourne le nombre d'éléments présents dans la pile <s>.  */

  unsigned stack_size(stack *s)
  {
    return s -> nb_values;
  }

  /*  Détermine si la pile <s> est vide.  */

  int stack_is_empty(stack *s)
  {
    return s -> nb_values == 0;
  }
```

stack.h:

```
  #define MAX_STACK_SIZE  1000

  typedef struct
  {
    void    *content[MAX_STACK_SIZE];
    unsigned nb_values;
  } stack;

  stack   *stack_new(void);
  void     stack_free(stack *);
  int      stack_push(stack *, void *);
  void    *stack_pop(stack *);
  void    *stack_top(stack *);
  unsigned stack_size(stack *);
  int      stack_is_empty(stack *);
```

FIGURE 7.4 – Implémentation d'une pile basée sur un vecteur (2/2)

```
#include <stdio.h>
#include "stack.h"

int main()
{
  int    i = 10, i2;
  double d = 3.1416, d2;
  stack *s;

  s = stack_new();
  if (!s || stack_push(s, &i) || stack_push(s, &d))
    return -1;

  d2 =  *((double *) stack_pop(s));
  i2 =  *((int *) stack_pop(s));

  printf("valeurs dépilées: %lf, %d.\n", d2, i2);

  stack_free(s);

  return 0;
}
```

FIGURE 7.5 – Programme de test de la structure de pile

Les détails de ce programme appellent peu de commentaires. En cas d'erreur, les fonctions `stack_new`, `stack_pop` et `stack_top` retournent un pointeur vide, et la fonction `stack_push` un entier non nul.

Un exemple de programme de test de cette structure de données est fourni à la figure 7.5. Ce programme alloue une pile par un appel à `stack_new`, et y empile ensuite la valeur entière 10 et la valeur réelle 3.1416. Ces opérations sont effectuées en passant à la fonction `stack_push` des pointeurs vers deux variables `i` et `d` contenant ces valeurs. Les situations d'erreur sont gérées grâce au mécanisme d'évaluation en circuit court de l'opérateur "`||`" : si l'appel à `stack_new` retourne le pointeur vide, ou si un des appels à `stack_push` retourne un entier non nul, alors la fonction `main` interrompt immédiatement son exécution et retourne le code de diagnostic -1. Sinon, le programme dépile deux valeurs qu'il place dans des variables `d2` et `i2`, dont il affiche ensuite le contenu. La dernière opération consiste à invoquer `stack_free` afin de libérer les ressources liées à la pile.

7.2.4 Application : appariement de parenthèses

Nous venons de voir ce qu'est une pile et comment l'implémenter, mais cela ne nous apprend pas quelle peut bien être l'utilité d'une telle structure de données. On utilise une pile dans les situations où l'on est amené à produire et à consommer des données, et où la nature du problème à résoudre fait que ces données sont toujours consommées dans l'ordre inverse de leur production.

Pour illustrer cela, nous considérons le problème qui consiste à déterminer si différentes formes de parenthèses (c'est-à-dire des parenthèses simples "`(`" et "`)`", des crochets "`[`" et "`]`" et des accolades "`{`" et "`}`") sont correctement appariées dans un texte fourni en entrée. Par exemple, la réponse à cette question est positive pour le texte

```
[- (b) + sqrt(4 * (a) * (c))] / (2 * a)
```

mais négative pour

```
((x) + (y)] / 2.
```

En effet, dans ce dernier texte, la première parenthèse ouvrante "`(`" est fermée par le symbole "`]`" au lieu d'une parenthèse fermante "`)`".

Une structure de pile est bien adaptée pour résoudre ce problème, car on ferme les parenthèses dans l'ordre inverse où on les ouvre. Plus précisément, lorsqu'on rencontre un symbole fermant "`)`", "`]`" ou "`}`", celui-ci se rapporte au symbole ouvrant "`(`", "`[`" ou "`{`" le plus récent qui n'a pas déjà été fermé. Une solution simple consiste donc à lire le texte caractère par caractère, et à maintenir une structure de pile qui contient à chaque instant les parenthèses ouvrantes qui ont été rencontrées et qui n'ont pas encore été fermées. On obtient l'algorithme suivant.

1. On crée une pile, initialement vide.
2. On examine les caractères du texte, dans l'ordre. Pour chaque caractère c :
   - Si c n'est pas une parenthèse, alors on l'ignore.
   - Si c est une parenthèse ouvrante, alors on l'empile.
   - Si c est une parenthèse fermante, alors :
     2.1 On dépile un caractère c'.
     2.2 Si cette opération échoue (car la pile est vide), ou si c' n'est pas le caractère symétrique de c, alors on signale une erreur.
3. S'il reste un ou plusieurs caractères dans la pile après avoir examiné la totalité du texte, alors on signale une erreur.

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

static void verifie_parenthese(stack *s, char c, unsigned ligne, unsigned col)
{
  char *p;

  p = stack_pop(s);
  if (!p || *p != c)
    printf("erreur: Symbole incorrect ligne %u, colonne %u.\n", ligne, col);
  free(p);
}

static void verifie_fichier(FILE *f)
{
  stack    *s;
  unsigned  num_ligne, num_col;
  char      c, *p;

  s = stack_new();
  if (!s)
    exit(-1);

  for (num_ligne = num_col = 1;; num_col++)
    {
      c = fgetc(f);  /* Lecture d'un caractère dans le fichier. */
      switch (c)
        {
        case EOF:    /* Fin du fichier. */
          if (!stack_is_empty(s))
            {
              printf("erreur: %d symbole(s) encore ouvert(s) en fin de fichier.\n",
                     stack_size(s));
              while (stack_size(s))
                free(stack_pop(s));
            }

          stack_free(s);
          return;

        case '\n':
          num_ligne++;
          num_col = 0;
          break;

        case '(':
        case '[':
        case '{':
          p = malloc(1);
          if (!p)
            exit(-1);

          *p = c;

          if (stack_push(s, p))
            exit(-1);
          break;
```

FIGURE 7.6 – Appariement de parenthèses (1/2)

```
        case ')':
          verifie_parenthese(s, '(', num_ligne, num_col);
          break;

        case ']':
          verifie_parenthese(s, '[', num_ligne, num_col);
          break;

        case '}':
          verifie_parenthese(s, '{', num_ligne, num_col);
          break;
        }
    }
}

int main(int argc, char *argv[])
{
  FILE *f;

  if (argc != 2)
    {
      printf("usage: %s <nom-fichier>\n", argv[0]);
      exit(-1);
    }

  f = fopen(argv[1], "r");  /* Ouverture d'un fichier en lecture. */
  if (!f)
    {
      printf("impossible d'ouvrir le fichier [%s]\n", argv[1]);
      exit(-1);
    }

  verifie_fichier(f);

  fclose(f);  /* Fermeture du fichier. */
  exit(0);
}
```

FIGURE 7.7 – Appariement de parenthèses (2/2)

Un programme implémentant cet algorithme est donné aux figures 7.6 et 7.7. Bien sûr, ce programme nécessite une implémentation de la structure de pile, et doit donc être compilé avec le fichier source `stack.c` des figures 7.3 et 7.4. Le programme inclut le fichier d'en-tête `stack.h` afin de disposer de la définition du type `stack` et du prototype des fonctions de manipulation d'une pile.

Pour pouvoir traiter n'importe quel texte, le programme lit celui-ci depuis un *fichier*, dont le nom lui est fourni par l'intermédiaire d'un argument d'exécution. Le programme commence donc par vérifier si le nombre d'arguments d'exécution, qui est égal à la valeur du paramètre `argc` de la fonction `main`, vaut 2, ce qui correspond à un seul argument d'exécution suivant le nom du programme. Dans le cas contraire, le programme affiche un message d'erreur rappelant comment l'utiliser correctement, et termine son exécution en retournant un code de diagnostic égal à -1.

La bibliothèque standard fournit des structures de données et des fonctions permettant de manipuler des fichiers. Les définitions et déclarations correspondantes se trouvent dans le fichier d'en-tête standard `stdio.h`. Pour ouvrir un fichier, une possibilité consiste à invoquer la fonction `fopen` de la bibliothèque standard, en lui passant comme arguments le nom de ce fichier et la chaîne constante "`r`" indiquant un mode de lecture [1]. Cette fonction retourne un pointeur vers une structure de données de la bibliothèque, de type `FILE`, représentant le fichier qui vient d'être ouvert, ou un pointeur vide dans le cas d'une erreur. Dans le programme, la fonction `fopen` est invoquée avec comme premier argument la chaîne de caractères présente dans `argv[1]`, qui correspond à l'argument d'exécution fourni au programme lors de son lancement. Après avoir ouvert le fichier, la fonction `main` invoque la fonction `verifie_fichier` chargée de vérifier si son contenu apparie correctement les parenthèses, et ferme ensuite le fichier (ce qui consiste à libérer les ressources qui lui sont liées) en invoquant la fonction `fclose` de la bibliothèque standard.

Le reste du programme implémente directement l'algorithme que nous avons développé. Pour lire le prochain caractère du fichier, on emploie la fonction `fgetc` de la bibliothèque standard ; celle-ci retourne un entier égal à la valeur numérique du caractère lu, ou à la constante `EOF` lorsque la fin du fichier est atteinte. Le programme maintient deux variables `num_ligne` et `num_colonne` qui contiennent à chaque instant le numéro de la ligne et de la colonne du caractère courant ; cela permet d'afficher en cas d'erreur d'appariement de parenthèses la position exacte du caractère problématique. La variable `num_colonne` est incrémentée à chaque lecture d'un nouveau caractère. Elle est remise à zéro, et la variable `num_ligne` est incrémentée, à chaque lecture d'un saut de ligne "`\n`". De cette façon, la valeur de `num_col` deviendra égale à 1 pour le caractère suivant, étant donné qu'elle est incrémentée à chaque itération de la boucle `for`.

Dans notre implémentation de la structure de pile, la fonction `stack_push` reçoit en argument un pointeur vers la donnée qui doit être empilée. Cette donnée correspond ici au symbole

1. D'autres valeurs de cette chaîne permettent notamment d'indiquer un mode d'écriture ("`w`") ou d'ajout ("`a`").

d'une parenthèse ouvrante "`(`", "`[`" ou "`{`". Lorsque le programme rencontre un tel symbole, il alloue un bloc de mémoire de taille 1 par un appel à la fonction `malloc`, recopie le caractère concerné dans ce bloc de mémoire, et transmet un pointeur vers ce bloc à `stack_push`. Réciproquement, lorsqu'une valeur est extraite de la pile grâce à `stack_pop`, elle est interprétée comme un pointeur vers un bloc, qui est libéré par `free` après avoir été lu. Quand la fin du fichier est atteinte, tous les blocs correspondant à des valeurs encore présentes sur la pile sont libérés, avant de libérer la pile elle-même.

7.2.5 Implémentation à l'aide d'une liste liée

La plupart des structures de données peuvent être implémentées de différentes façons, parfois basées sur des principes très différents, et la structure de pile ne fait pas exception. La solution développée à la section 7.2.3 a l'avantage d'être simple, mais présente l'inconvénient de devoir estimer a priori le nombre d'éléments à allouer pour le vecteur représentant le contenu de la pile. Cela peut conduire à gaspiller de la mémoire si ce nombre est surestimé, ou à atteindre prématurément la limite de capacité de la structure s'il est sous-estimé. Nous présentons ici une autre solution qui ne possède pas cet inconvénient. L'idée est que cette implémentation alternative présente exactement la même interface que la précédente, afin de pouvoir l'utiliser dans une application telle que celle des figures 7.6 et 7.7 sans devoir en modifier le code source. Il s'agit d'une illustration d'un mécanisme de *modularité* : il est possible de modifier l'implémentation d'une partie d'un programme sans que cela n'affecte le reste du projet, à condition d'en préserver l'interface. Ce mécanisme est essentiel pour le développement de logiciels de grande envergure.

Nous avons déjà élaboré à la section 6.1.3 un algorithme capable de lire un ensemble de valeurs et de les restituer en ordre inverse, sans devoir connaître a priori le nombre de ces valeurs. Cet algorithme reposait sur la construction d'une *liste simplement liée* dont chaque élément représentait une valeur. La lecture d'une valeur s'effectuait en créant un nouvel élément placé en tête de la liste, et la restitution d'une valeur consistait à retirer l'élément situé en tête de liste. Le même principe permet d'implémenter les opérations `push` et `pop` d'une pile. L'avantage de cette approche est que les éléments de la liste liée peuvent être alloués dynamiquement en fonction des besoins. L'inconvénient d'une liste liée est que l'opération qui consiste à compter le nombre d'éléments qu'elle contient est inefficace, car elle demande de la parcourir entièrement. Une solution permettant de pallier cet inconvénient consiste à prévoir un champ supplémentaire pour un compteur du nombre d'éléments présents dans la liste à chaque instant.

Une illustration de cette structure de pile basée sur une liste simplement liée est donnée à la figure 7.8, pour une pile contenant les valeurs 10, 20 et 30 empilées dans cet ordre. Elle est composée d'un champ `first` pointant vers la tête de la liste liée, et un champ `nb_values` retenant le nombre d'éléments courant de la pile. Une implémentation de cette structure est fournie aux figures 7.9, 7.10 et 7.11. Comme pour l'implémentation basée sur un vecteur développée à la section 7.2.3, les éléments de la pile sont représentés par des pointeurs vers des données de n'importe quel type.

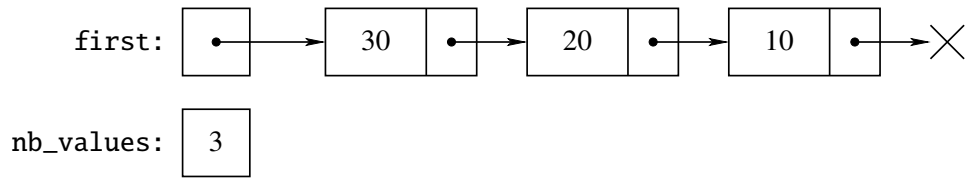


FIGURE 7.8 – Représentation d'une pile basée sur une liste liée

7.3 Les files

Nous abordons à présent l'étude d'une nouvelle structure de donnée, la *file*, ou *file d'attente (queue, FIFO buffer)*.

7.3.1 Définition

La file possède plusieurs points communs avec la structure de pile introduite à la section 7.2. Une file est une structure de données permettant de retenir une collection de valeurs, en disposant d'opérations permettant d'y ajouter ou retirer une valeur. À la différence d'une pile, l'accès aux éléments d'une file s'effectue selon une politique FIFO (*First In First Out*). Cela signifie que les éléments d'une file ne peuvent être extraits que dans l'ordre où ils y ont été placés : l'opération d'extraction retourne toujours l'élément qui, parmi ceux présents dans la file, s'y trouve depuis le plus longtemps. On peut établir une analogie avec une file d'attente pour accéder à un guichet ou au comptoir d'un magasin : les personnes qui quittent cette file pour être servies le font dans l'ordre où elles sont arrivées et se sont ajoutées à la queue. En informatique, les files d'attente sont notamment utilisées pour gérer les situations où l'on reçoit des requêtes pour accéder à une ressource partagée, par exemple une imprimante, et où ces requêtes doivent être traitées dans l'ordre où elles arrivent.

7.3.2 Interface

L'interface d'une file q est composée des fonctions suivantes. Tout comme pour le cas d'une pile, nous considérons que les valeurs v placées dans la file peuvent être de n'importe quel type.

— `send(`q`,` v`)` : Ajoute la valeur v à la collection des valeurs retenues par la file q.

— `receive(`q`)` : Extrait une valeur de la file q et retourne cette valeur. Selon la politique d'accès FIFO, la valeur extraite est, parmi celles présentes dans la file, celle qui a été ajoutée en premier lieu. Dans la cas où la pile est vide, cette opération signale une erreur.

stack.c:

```
#include <stdlib.h>
#include "stack.h"

/*  Retourne une nouvelle pile vide, ou NULL en cas d'erreur.  */

stack *stack_new()
{
  stack *s;

  s = malloc(sizeof(stack));
  if (!s)
    return NULL;

  s -> first = NULL;
  s -> nb_values = 0;

  return s;
}

/*  Libère la pile <s>.  */

void stack_free(stack *s)
{
  stack_element *e, *e_suivant;
  for (e = s -> first; e; e = e_suivant)
    {
      e_suivant = e -> suivant;
      free(e);
    }

  free(s);
}

/*  Empile le pointeur <p> sur la pile <s>.  Retourne 0 en cas de succès,
    ou -1 en cas d'insuffisance mémoire.  */

int stack_push(stack *s, void *p)
{
  stack_element *e;

  e = malloc(sizeof(stack_element));
  if (!e)
    return -1;

  e -> content = p;
  e -> next = s -> first;
  s -> first = e;
  s -> nb_values++;

  return 0;
}
```

FIGURE 7.9 – Implémentation d'une pile basée sur une liste liée (1/3)

```
/*  Dépile un pointeur depuis la pile <s> et retourne ce pointeur,
    ou NULL si la pile est vide.  */

void *stack_pop(stack *s)
{
  stack_element *e;
  void          *p;

  if (!(s -> first))
    return NULL;

  e = s -> first;
  p = e -> content;

  s -> first = e -> next;
  s -> nb_values--;

  free(e);

  return p;
}

/*  Retourne le pointeur au sommet de la pile <s>, sans le dépiler,
    ou NULL si la pile est vide.  */

void *stack_top(stack *s)
{
  if (!(s -> first))
    return NULL;

  return s -> first -> content;
}

/*  Retourne le nombre d'éléments présents dans la pile <s>.  */

unsigned stack_size(stack *s)
{
  return s -> nb_values;
}

/*  Détermine si la pile <s> est vide.  */

int stack_is_empty(stack *s)
{
  return s -> nb_values == 0;
}
```

FIGURE 7.10 – Implémentation d'une pile basée sur une liste liée (2/3)

stack.h:

```
  typedef struct _stack_element
  {
    void                  *content;
    struct _stack_element *next;
  } stack_element;

  typedef struct
  {
    stack_element *first;
    unsigned       nb_values;
  } stack;

  stack   *stack_new(void);
  void     stack_free(stack *);
  int      stack_push(stack *, void *);
  void    *stack_pop(stack *);
  void    *stack_top(stack *);
  unsigned stack_size(stack *);
  int      stack_is_empty(stack *);
```

FIGURE 7.11 – Implémentation d'une pile basée sur une liste liée (3/3)

FIGURE 7.12 – Exemple de contenus de file

— `front(`q`)` : Retourne la valeur située en *tête* de la file q, c'est-à-dire celle qui a été ajoutée en premier lieu parmi celles présentes dans la file, sans l'extraire. Cette opération ne modifie donc pas le contenu de la file. Elle signale une erreur si elle est appliquée à une file vide.

— `size(`q`)` : Retourne le nombre de valeurs contenues dans la file q.

— `is_empty(`q`)` : Retourne une valeur booléenne indiquant si la file q est vide.

Nous illustrons ces opérations à l'aide de la séquence suivante, que l'on applique à une file q supposée initialement vide.

```
send(q, 10)
send(q, 20)
receive(q)
send(q, 30)
receive(q)
receive(q)
```

Les deux premières opérations ajoutent successivement les valeurs 10 et 20 à la file ; après les avoir effectuées, le contenu de la file est celui de la figure 7.12 (a), où la tête de la file est située à gauche. L'opération suivante extrait une valeur de la file, et retourne donc celle qui a été ajoutée en premier lieu, c'est-à-dire 10. Ensuite, on ajoute la valeur 30, ce qui mène au contenu de la file montré à la figure 7.12 (b). Les deux dernières opérations extraient les valeurs encore présentes dans la file dans l'ordre où elles y ont été ajoutées, et produisent donc successivement les valeurs 20 et 30.

7.3.3 Implémentation à l'aide d'un vecteur

Tout comme pour une pile, on peut implémenter une file en plaçant les valeurs qu'elle contient dans les éléments consécutifs d'un vecteur de taille donnée. Cette taille correspondra à la capacité maximale de la file. La façon de manipuler ce vecteur est cependant plus compliquée pour une file que pour une pile : pour une pile, les opérations `push` et `pop` peuvent être

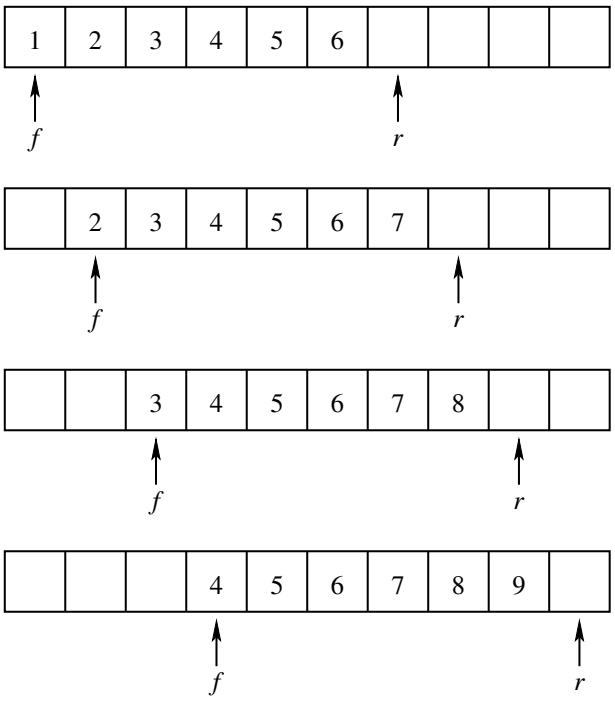


FIGURE 7.13 – Index repérant les éléments d'une file dans un vecteur

implémentées en modifiant uniquement le sommet de la pile, sans que les valeurs situées à la base de celle-ci ne soient affectées. Pour une file, la situation est différente : une opération `receive` modifie la tête de la file, et une opération `send` son autre extrémité.

Si l'on considère le cas particulier d'une file contenant un certain nombre n de valeurs, à laquelle on applique répétitivement une opération `send` suivie d'une opération `receive`, on souhaite éviter de devoir décaler les n éléments du vecteur qui représentent le contenu de la file, ce qui est inefficace si n est grand. Une solution consiste à maintenir deux indices f ("*front*") et r ("*rear*") vers les éléments du vecteur, définis de la façon suivante. L'indice f repère la valeur la plus ancienne parmi celles contenues dans la file, c'est-à-dire celle qui a été ajoutée en premier lieu. L'indice r repère l'élément du vecteur qui suit la valeur la plus récente, c'est-à-dire l'emplacement où sera écrite la prochaine valeur que l'on ajoutera à la file. Les opérations `send` et `receive` s'effectuent alors en déplaçant ces indices : `send` déplace r d'une position vers la droite et `receive` déplace f d'une position vers la droite. Ce mécanisme est illustré à la figure 7.13, qui montre l'effet d'une opération `send` suivie d'une opération `receive` appliquées répétitivement à une file de capacité 10 contenant initialement 6 valeurs. Dans cet exemple, les valeurs ajoutées par `send` sont croissantes.

Avec cette approche, les valeurs contenues dans la file sont représentées par les éléments

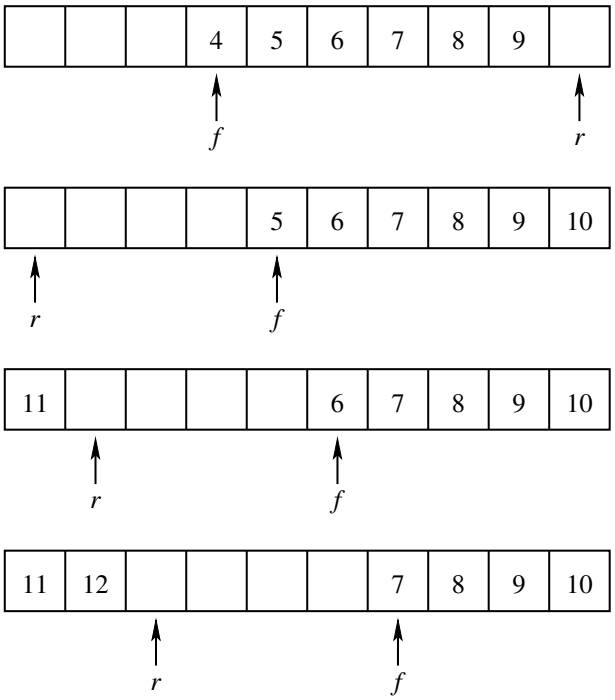


FIGURE 7.14 – Mécanisme de tableau circulaire

du vecteur qui commencent à l'indice f et se terminent immédiatement avant l'indice r. Un problème est que les indices f et r se déplacent toujours vers la droite lorsqu'on effectue des opérations `receive` et `send`, ce qui peut les conduire à sortir du vecteur. On résout ce problème en considérant que les éléments du vecteur sont organisés de façon *circulaire*. Pour un vecteur de taille m, possédant donc des éléments indicés de 0 à $m - 1$, cela signifie que si un indice devient égal à m, alors on le ramène à 0. En d'autres termes, on considère que l'élément d'indice $m - 1$ est suivi par l'élement d'indice 0. Ce mécanisme est illustré à la figure 7.14, qui montre la suite de la séquence d'opérations entamée à la figure 7.13.

Il reste une difficulté à résoudre. La situation particulière où l'on a $f = r$ est ambiguë : elle correspond à la fois au cas d'une file vide, c'est-à-dire ne contenant aucune valeur, mais aussi à celui d'une file saturée, c'est-à-dire contenant un nombre de valeurs égal à sa capacité. Par exemple, on arrive à cette situation en appliquant 4 opérations `send` ou 6 opérations `receive` à n'importe laquelle des configurations des figures 7.13 et 7.14.

Il est cependant indispensable de pouvoir distinguer le cas d'une file vide de celui d'une file saturée. En particulier, une opération `send` doit retourner une erreur dans le deuxième cas mais pas dans le premier, au contraire d'une opération `receive` qui ne doit signaler une erreur que dans le premier cas. L'opération `size` nécessite aussi de pouvoir déterminer le nombre de valeurs contenues dans la pile.

Ce problème admet deux solutions simples. La première consiste à retenir séparément le nombre d'éléments contenus dans la file, comme nous l'avons fait à la section 7.2.3 pour la structure de pile. Une autre stratégie consiste à interdire d'utiliser tous les éléments du vecteur ; pour un vecteur de taille m, cela revient à considérer que la file est saturée dès qu'elle contient $m - 1$ valeurs.

Une implémentation basée sur cette seconde solution est données aux figures 7.15 et 7.16. De façon similaire à la structure de pile développée à la section 7.2.3, les valeurs contenues dans la file sont de type `void *`. La fonction `queue_send` vérifie que la file n'est pas saturée, et écrit ensuite la nouvelle valeur à l'endroit désigné par le champ `rear`. Ce champ est ensuite incrémenté de façon circulaire. De même, la fonction `queue_receive` commence par tester si la file est vide, ce qui se produit lorsque les champs `front` et `rear` sont égaux. Elle lit ensuite la valeur désignée par `front`, et incrémente ce champ de façon circulaire. La fonction `queue_size` détermine le nombre de valeurs contenues dans la file en calculant la différence entre `rear` et `front`, modulo la taille `MAX_QUEUE_SIZE` du vecteur. Remarquons que l'on évalue

```
(MAX_QUEUE_SIZE + q -> rear - q -> front) % MAX_QUEUE_SIZE
```

et non

```
q -> rear - q -> front) % MAX_QUEUE_SIZE
```

.

Cela permet de garantir que la première opérande de l'opérateur "`%`" possède une valeur positive ou nulle, ce qui est nécessaire pour que cet opérateur corresponde à la notion mathématique

queue.c:

```
#include <stdlib.h>
#include "queue.h"

/*  Retourne une nouvelle file vide, ou NULL en cas d'erreur.  */

queue *queue_new()
{
  queue *q;

  q = malloc(sizeof(queue));
  if (!q)
    return NULL;

  q -> front = 0;
  q -> rear = 0;

  return q;
}

/*  Libère la file <q>.  */

void queue_free(queue *q)
{
  free(q);
}

/*  Ajoute le pointeur <p> à la file <q>.  Retourne 0 en cas de succès,
    ou -1 si la file est saturée.  */

int queue_send(queue *q, void *p)
{
  if (queue_size(q) >= MAX_QUEUE_SIZE - 1)
    return -1;

  q -> contents[q -> rear] = p;
  q -> rear = (q -> rear + 1) % MAX_QUEUE_SIZE;

  return 0;
}

/*  Retire un pointeur depuis la file <q> et retourne ce pointeur,
    ou NULL si la file est vide.  */

void *queue_receive(queue *q)
{
  void *p;

  if (q -> front == q -> rear)
    return NULL;

  p = q -> contents[q -> front];
  q -> front = (q -> front + 1) % MAX_QUEUE_SIZE;

  return p;
}
```

FIGURE 7.15 – Implémentation d'une file basée sur un vecteur (1/2)

```
/*  Retourne le pointeur en tête de la file <q>, sans le retirer,
    ou NULL si la file est vide.  */

void *queue_front(queue *q)
{
  if (q -> front == q -> rear)
    return NULL;

  return q -> contents[q -> front];
}

/*  Retourne le nombre d'éléments présents dans la file <q>.  */

unsigned queue_size(queue *q)
{
  return (MAX_QUEUE_SIZE + q -> rear - q -> front) % MAX_QUEUE_SIZE;
}

/*  Détermine si la file <q> est vide.  */

int queue_is_empty(queue *q)
{
  return q -> front == q -> rear;
}
```

queue.h:

```
#define MAX_QUEUE_SIZE  1000

typedef struct
{
  void    *contents[MAX_QUEUE_SIZE];
  unsigned front, rear;
} queue;

queue   *queue_new(void);
void     queue_free(queue *);
int      queue_send(queue *, void *);
void    *queue_receive(queue *);
void    *queue_front(queue *);
unsigned queue_size(queue *);
int      queue_is_empty(queue *);
```

FIGURE 7.16 – Implémentation d'une file basée sur un vecteur (2/2)

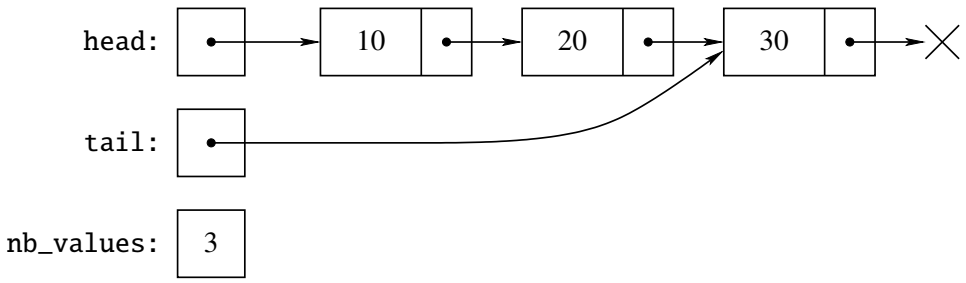


FIGURE 7.17 – Représentation d'une file basée sur une liste liée

de modulo, comme nous l'avons vu à la section 2.3.1.

7.3.4 Implémentation à l'aide d'une liste liée

On peut chercher, comme dans le cas de la structure de pile, à développer une implémentation de la file qui serait plus dynamique, c'est-à-dire qui serait capable d'adapter sa capacité aux besoins de l'application pendant l'exécution du programme. Pour la pile, nous avons obtenu à la section 7.2.5 une implémentation basée sur la notion de liste simplement liée. Nous montrons à présent comment implémenter une structure de file à l'aide d'une telle liste liée.

L'idée consiste à représenter le contenu d'une file par une liste liée dont le premier élément correspond à la tête de la file, et où chaque élément suit celui qui a été ajouté immédiatement avant lui. On peut alors effectuer une opération `send` en ajoutant un nouvel élément à la fin de la liste, et une opération `receive` en retirant l'élément situé en tête de liste. La seule difficulté technique est celle de trouver la fin de la liste. Pour éviter de devoir résoudre ce problème en parcourant tous les éléments de la liste, ce qui est inefficace, on gère la liste en maintenant deux pointeurs : un pointeur `head` vers son premier élément, et un pointeur `tail` vers son dernier élément. Dans le cas particulier où la liste est vide, les pointeurs `head` et `tail` sont tous deux vides. Enfin, on définit un champ supplémentaire retenant le nombre de valeurs contenues dans la file, encore une fois dans le but d'éviter de devoir compter explicitement le nombre d'éléments de la liste liée. La figure 7.17 illustre cette structure de données dans le cas d'une file contenant les valeurs [2] 10, 20 et 30, ajoutées dans cet ordre.

Une implémentation basée sur ces principes est donnée aux figures 7.18, 7.19 et 7.20. La fonction `queue_send` alloue un nouvel élément *e* destiné à être placé à la fin de la liste liée. Son

2. Nous montrons ici le cas d'une file contenant des entiers, dans un but de simplicité. Comme cela a déjà été expliqué, notre implémentation définira des files plus générales, dont les valeurs seront de type `void *`.

queue.c:

```
#include <stdlib.h>
#include "queue.h"

/*  Retourne une nouvelle file vide, ou NULL en cas d'erreur.  */

queue *queue_new()
{
  queue *q;

  q = malloc(sizeof(queue));
  if (!q)
    return NULL;

  q -> head = NULL;
  q -> tail = NULL;
  q -> nb_values = 0;

  return q;
}

/*  Libère la file <q>.  */

void queue_free(queue *q)
{
  queue_element *e, *e_next;
  for (e = q -> head; e; e = e_next)
    {
      e_next = e -> next;
      free(e);
    }

  free(q);
}

/*  Ajoute le pointeur <p> à la file <q>.  Retourne 0 en cas de succès,
    ou -1 en cas d'insuffisance mémoire.  */

int queue_send(queue *q, void *p)
{
  queue_element *e;

  e = malloc(sizeof(queue_element));
  if (!e)
    return -1;

  e -> content = p;
  e -> next = NULL;

  if (q -> tail)
    q -> tail -> next = e;
  else
    q -> head = e;

  q -> tail = e;
  q -> nb_values++;

  return 0;
}
```

FIGURE 7.18 – Implémentation d'une file basée sur une liste liée (1/3)

```
/*  Extrait un pointeur de la file <q> et retourne ce pointeur,
    ou NULL si la file est vide.  */

void *queue_receive(queue *q)
{
  queue_element *e;
  void          *p;

  if (!(q -> head))
    return NULL;

  e = q -> head;
  p = e -> content;

  q -> head = e -> next;
  if (!--(q -> nb_values))
    q -> tail = NULL;

  free(e);

  return p;
}

/*  Retourne le pointeur en tête de la file <q>, sans le retirer,
    ou NULL si la file est vide.  */

void *queue_front(queue *q)
{
  if (!(q -> head))
    return NULL;

  return q -> head -> content;
}

/*  Retourne le nombre d'éléments présents dans la file <q>.  */

unsigned queue_size(queue *q)
{
  return q -> nb_values;
}

/*  Détermine si la file <q> est vide.  */

int queue_is_empty(queue *q)
{
  return q -> nb_values == 0;
}
```

FIGURE 7.19 – Implémentation d'une file basée sur une liste liée (2/3)

queue.h:

```
typedef struct _queue_element
{
  void                  *content;
  struct _queue_element *next;
} queue_element;

typedef struct
{
  queue_element *head, *tail;
  unsigned       nb_values;
} queue;

queue   *queue_new(void);
void     queue_free(queue *);
int      queue_send(queue *, void *);
void    *queue_receive(queue *);
void    *queue_front(queue *);
unsigned queue_size(queue *);
int      queue_is_empty(queue *);
```

FIGURE 7.20 – Implémentation d'une file basée sur une liste liée (3/3)

champ `next` est donc initialisé à l'aide d'un pointeur vide. Il y a deux cas à considérer : si la file est vide, alors les champs `head` et `tail` doivent tous deux pointer vers *e*. Si la file n'est pas vide, alors le champ `next` de l'élément pointé par `tail` doit maintenant pointer vers *e*. Ensuite, `tail` doit être modifié pour pointer également vers *e*.

La fonction `queue_receive` doit retirer de la liste l'élément pointé par `head`, ce qui implique que `head` doit devenir égal à la valeur du champ `next` de cet élément. Un cas particulier doit être traité : si la file devient vide après l'opération, alors `head` prendra pour valeur le pointeur vide, mais il faut également veiller à ce que `tail` prenne aussi cette valeur. On peut indifféremment tester si la file est vide en détérminant si `head` est le pointeur vide, ou si `tail` l'est, ou si la valeur courante de `nb_values` est nulle.

7.4 Exercices

Les exercices de cette section sont difficiles et ne font pas partie de la matière du cours d'*introduction à l'informatique*. Leur intérêt est de montrer quelques applications des structures de données développées dans ce chapitre.

1. Un *arbre binaire* est une structure de données composée d'un certain nombre de *nœuds*. Chaque nœud retient une valeur, ainsi que deux pointeurs vers des nœuds appelés *fils gauche* et *fils droit*. Chacun de ces pointeurs peut être vide. Si l'arbre n'est pas vide, alors il possède un nœud particulier appelé sa *racine*, qui est le seul à ne pas être le fils (gauche ou droit) d'un nœud. À partir de la racine, il doit être possible d'atteindre tous les nœuds de l'arbre en suivant une certaine succession de pointeurs vers le fils gauche ou le fils droit du nœud courant.

   (a) Écrire en langage C une bibliothèque permettant de représenter et de manipuler des arbres binaires retenant des valeurs entières. Il doit être possible de créer un arbre composé d'une racine *r* retenant un entier donné. Les fils gauche et droit de *r* doivent correspondre à la racine de deux arbres donnés, chacun d'entre eux pouvant être vide. On suppose que ces deux arbres contiennent des nœuds distincts, et distincts de *r*.

   (b) Écrire une fonction récursive permettant d'afficher toutes les valeurs contenues dans les nœuds d'un arbre binaire donné. Pour chaque nœud *n*, les valeurs situées dans les nœuds accessibles via son fils gauche doivent être affichées avant celle associée à *n*. Les valeurs retenues dans les nœuds accessibles par l'intermédiaire du fils droit de *n* doivent être affichées après celle de *n*.

   (c) Écrire une fonction itérative qui effectue le même travail que la fonction obtenue au point (a). *Suggestion :* Utiliser une pile pour retenir les nœuds dont les successeurs doivent encore être explorés.

2. Un *graphe dirigé* est une structure de données qui, comme l'arbre binaire, est composée d'un certain nombre de *nœuds* retenant chacun une valeur. De chaque nœud sont issus un certain nombre d'*arcs* qui mènent chacun vers un nœud du graphe. S'il n'est pas vide, un graphe possède un nœud particulier appelé son nœud initial.

   (a) Écrire en langage C une bibliothèque permettant de représenter et de manipuler des graphes dirigés retenant des valeurs entières. Il doit être possible

   — de créer un graphe vide.

   — d'ajouter un nouveau nœud à un graphe donné, contenant une valeur donnée. Le premier nœud ainsi créé devient le nœud initial.

   — d'ajouter un arc dans un graphe donné, depuis un nœud donné vers un nœud donné.

   Suggestion : Utiliser des listes simplement liées pour représenter l'ensemble des nœuds d'un graphe, ainsi que l'ensemble des arcs issus de chaque nœud.

   (b) Un nœud d'un graphe est dit *accessible* s'il peut être atteint depuis le nœud initial en suivant un certain nombre d'arcs.

   Écrire une fonction qui affiche la valeur retenue dans chaque nœud accessible d'un graphe donné, dans un ordre arbitraire.

   Suggestions :

   — Prévoir dans chaque nœud un champ retenant si ce nœud a déjà été visité ou non.

   — Utiliser une pile ou une file contenant les nœuds dont les successeurs doivent encore être explorés.