SCIENCES SORBONNE UNIVERSITÉ

Elements de programmation 2
UE LU1IN002. Poly de cours.

Auteur: M. Carpentier, C. Dutheillet, P. Manoury, à partir du poly de S. Doncieux
Date: 7 avril 2020

Ce poly a été écrit très rapidement dans un contexte difficile. Nous nous excusons des coquilles qui y sont, elle seront corrigées au fur et à mesure.

1 COURS 1 - DE PYTHON À C : LA SYNTAXE DU LANGAGE C

Cours assuré en cours magistral. Poly à venir... Si besoin lire https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1042_les-bases-du-langage-c/, la partie I.

2 COURS 2 – ORDINATEURS ET PROGRAMME

Cours assuré en cours magistral. Poly à venir...

3 COURS 3 - TABLEAUX, POINTEURS ET ALLOCATION

Cours assuré en cours magistral. Poly à venir... Si besoin, lire https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4277_les-pointeurs/,https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4281_les-tableaux/ et https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4285_lallocation-dynamique/

4 COURS 4 – ALGORITHMES

Cours assuré en cours magistral. Poly à venir...

5 COURS 5 - ARITHMÉTIQUE DES POINTEURS ET CHAÎNES DE CARACTÈRES

A Cours assuré en cours magistral. Poly à venir..... Si besoin lire https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4283_les-chaines-de-caracteres/

6 COURS 6 - STRUCTURES

Cours assuré en cours magistral. Poly à venir... Si besoin, lire https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4279_structures/

7 COURS 7 À 9 – LISTES CHAÎNÉES

Ce chapitre présente le principe des listes dites chaînées et leur implantation en langage C. Ce type de structure est très courant en informatique et son implantation en C est l'occasion d'utiliser de façon plus approfondie les structures et les pointeurs qui ont été vus auparavant. Il est indispensable d'avoir bien compris ces deux notions avant d'aborder celle-ci.

7.1 INTRODUCTION

7.1.1 Des limitations des tableaux

Les tableaux permettent de stocker un nombre fixé à l'avance de variables d'un type donné. Par exemple, un tableau déclaré de la façon suivante : `int tab[10]` permet de stocker 10 variables de type `int`, stockées les unes après les autres.

Les tableaux sont plutôt simples à définir et à utiliser, cependant, ils disposent de quelques inconvénients. Tout d'abord, la taille, qui est imposée à la déclaration (ou à l'allocation) peut être un problème. Si on souhaite agrandir un tableau, il faut allouer un tableau de taille supérieure, recopier le contenu de l'ancien tableau dans le nouveau, puis libérer la mémoire allouée à l'ancien tableau [1]. Les limitations des tableaux sont encore plus flagrantes si l'on souhaite ajouter un élément au milieu du tableau : il va falloir décaler tous les éléments qui seront après le nouvel élément (cf. figure **??**).

7.1.2 Idée intuitive

Les contraintes que l'on a identifiées sont liées au fait que toutes les cases d'un tableau sont stockées les unes à la suite des autres. Cela impose de disposer de suffisamment d'espace contigu et implique de déplacer potentiellement de grandes quantités de données si l'on souhaite insérer de nouvelles valeurs dans le tableau. Pourquoi ne pas définir une nouvelle structure dans laquelle les éléments ne se suivent pas les uns après les autres ? Dans le cas d'un tableau, l'adresse de l'élément qui suit une case particulière est obtenue en ajoutant `1` au pointeur sur cette case. Si l'on ne souhaite plus stocker les cases de ce nouveau type de tableau les unes à la suite des

1. La fonction `realloc` effectue toutes ces opérations

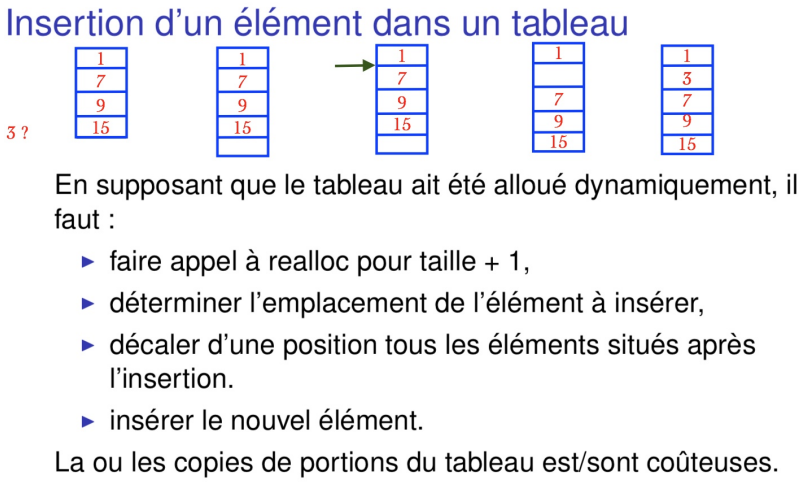


FIGURE 1 – Insertion de 3 dans un tableau d'entiers triés par ordre croissant.

autres, il faut se donner un moyen de les retrouver. Pourquoi ne pas indiquer, dans chaque case, l'emplacement de la case suivante ? De cette façon, en partant de la case initiale, on pourra retrouver toutes les cases de ce "tableau". C'est le principe des listes chaînées. Chaque élément contient une donnée et l'adresse de l'élément suivant.

Qu'est-ce que cela implique ? Les différents éléments d'une liste chaînée n'ont pas besoin d'être les uns à la suite des autres. On peut les allouer séparément et stocker l'adresse mémoire d'un élément dans l'élément qui le précède pour pouvoir le retrouver. Lorsque l'on souhaite ajouter un élément à une telle liste, il suffit donc d'allouer un élément et un seul, et de mettre à jour la "chaîne". Pour mettre à jour la chaîne, il suffit de modifier les éléments qui indiquent l'emplacement de l'élément à ajouter, donc a priori uniquement l'élément qui le précède[2]. Ce qu'il faut noter ici, c'est que cette opération a un coût qui ne dépend pas de la taille de la liste. On a donc résolu ce qui nous posait problème.

Quelle conséquence a l'utilisation de listes chaînées ? Si l'on a réussi à résoudre des problèmes liés aux tableaux, il faut se demander si on a introduit d'autres problèmes. Si ce n'était pas le cas, on pourrait en effet remplacer systématiquement les tableaux par des listes chaînées sans se poser plus de questions. Avec les listes chaînées, on introduit en fait deux problèmes potentiels. Le premier est que l'on ne peut plus accéder directement à un élément de la liste. Dans le cas d'un tableau, on sait que pour trouver le 4eme élément d'un tableau `tab`, il suffit d'écrire `tab[3]` ou, écrit avec des pointeurs, `*(tab+3)` (le premier élément d'un tableau est `tab[0]`). L'ac-

2. Nous verrons plus tard des listes doublement chaînées qui contiennent à la fois l'adresse de l'élément suivant, et celle de l'élément précédent. Dans ce cas, il faudra aussi modifier l'élément suivant.

cès à un élément est donc immédiat et indépendant de la taille du tableau (il suffit de faire l'addition d'un entier à un pointeur). Dans le cas des listes chaînées, on ne connait pas à l'avance l'emplacement d'un élément : il est uniquement spécifié dans l'élément qui le précède. Il est donc nécessaire de parcourir la liste depuis son premier élément jusqu'à trouver l'élément cherché. Cette opération n'est plus indépendante du nombre d'éléments stockés dans la liste ! De plus, chaque élément doit contenir l'adresse mémoire de l'élément suivant, ce qui n'était pas utile avec un tableau. Une liste chaînée va donc occuper plus de mémoire qu'un tableau.

Quand choisir d'utiliser une liste chaînée plutôt qu'un tableau ? Nous avons vu que cette structure dispose d'avantages, mais aussi d'inconvénients. Elle sera donc à utiliser dans les cas où il faut insérer ou supprimer fréquemment des éléments. Le choix d'une telle structure ne saurait donc être systématique et nécessite donc de bien réfléchir à l'usage qui en sera fait. Tout ce que l'on peut faire avec une liste chaînée pourrait être fait avec un tableau. Seules des raisons d'efficacité peuvent pousser à utiliser de préférence l'un ou l'autre.

7.2 DÉFINITION D'UNE LISTE CHAÎNÉE EN C

Une liste chaînée est définie par une succession d'éléments contenant une donnée et l'adresse mémoire du prochain élément de la liste. Le type à définir pour caractériser une liste chaînée est donc uniquement le type d'un de ses éléments. Ce type respecte le schéma suivant :

```
1 struct _un_element{
2
3    /* champs de donnees */
4
5    struct _un_element *suivant;
6 };
```

Les données stockées peuvent être de n'importe quel type et peuvent contenir autant de champs que nécessaire. Le seul impératif est l'existence d'un champ de type pointeur sur la structure ainsi définie (son nom, de même que le nom choisi pour la structure, importe peu, nous utiliserons le nom `suivant` dans la suite de ce cours). Pour faciliter l'utilisation de cette structure, nous définirons simultanément un type synonyme de la façon suivante :

```
1  struct  _un_element{
2
3    /* champ(s) de donnees */
4
5    struct _un_element *suivant;
```

```
6 } ;
7 typedef struct _un_element Un_element;
```

Il est à remarquer que le type `Un_element` ne peut pas être utilisé dans la définition du champ `suivant` car il n'est pas encore connu par le compilateur lors de la la lecture de la structure. Par contre, il suffit de déplacer la ligne commençant par **`typedef`** et de la mettre avant la définition de la structure pour pouvoir utiliser le nom `Un_element` :

```
1 typedef struct _un_element Un_element;
2  struct  _un_element{
3
4    /* champ(s) de donnees */
5
6    Un_element *suivant;
7 } ;
```

Une liste chaînée est ensuite simplement définie comme un pointeur sur un de ces éléments. En pratique, la liste chaînée pointera sur le premier élément qu'elle contient. Une liste chaînée ne nécessite donc pas la définition d'un autre type, il suffit de déclarer un pointeur :

```
1 Un_element  *ma_liste=NULL;
```

La variable `ma_liste` déclarée ci-dessus contient la valeur `NULL`, autrement dit elle ne pointe sur aucun élément. C'est de cette façon que les **listes vides** seront représentées.

Remarque : rien ne distingue une liste chaînée d'un pointeur sur un élément, c'est l'usage qui sera fait de ce pointeur qui en fera une liste chaînée (de même qu'un `char` est un entier et que l'usage qui en est fait permet de le considérer soit comme un entier, soit comme un caractère).

Exemple 1, liste chaînée contenant des entiers :

```
1 typedef struct _cellule_t cellule_t;
2 struct _cellule_t{
3   int donnee;
4   cellule_t *suivant;
5 };
```

Exemple 2, liste chaînée contenant des chaînes de caractères (de 29 lettres au maximum) :

```
1 typedef struct _un_element_str Un_element_str
2 struct _un_element_str{
3   char mot[30];
4   Un_element_str *suivant;
```

```
5 };
```

Exemple 3, liste chaînée contenant un entier, un nombre réel et un mot :

```
1 typedef struct _un_element_divers Un_element_divers;
2 struct _un_element_divers{
3   int n;
4   float x;
5   char m[30];
6   Un_element_divers *suivant;
7 };
```

7.3 FONCTIONS DE MANIPULATION DES LISTES CHAÎNÉES

La manipulation des listes chaînées est un peu plus complexe que celle des tableaux. Pour cette raison, les opérations les plus courantes sont habituellement regroupées dans des fonctions.

Dans cette section seront présentées les fonctions implantant les opérations les plus courantes. Nous allons vous présenter d'autres types de parcours parcours dont voici la liste :

- Construire une liste
- Parcourir une liste :
  - Parcours simples :
    - Afficher une liste
    - Fonction retournant un entier : retourner la somme des éléments d'une liste
    - Fonction retournant une liste : retourner une nouvelle liste contenant les carrés des éléments
    - Fonction retournant une liste : retourner la liste ordonnée en sens inverse
  - Parcours pouvant être «interrompus»
    - Rechercher un élément dans une liste
  - Parcours nécessitant de conserver l'élément précédent :
    - Libérer un liste
    - Supprimer un élément d'une liste
    - Concaténer deux listes
  - Autres parcours

- Parcourir deux listes $->$ Fusion de deux listes ordonnées

Nous allons vous donner le code pour ces différents parcours, en version itérative et ensuite en version récursive (chapitre suivant).

Dans la suite de cette section, la structure utilisée pour un élément sera la suivante (la même que celle du TD 8) :

```
1  typedef struct _cellule_t cellule_t;
2  struct _cellule_t{
3      int donnee;
4      cellule_t *suivant;
5  };
```

Il ne s'agit bien sûr que d'un cas particulier de liste chaînée. Les fonctions données dans la suite de cette section devront donc être adaptées aux besoins en changeant le traitement réalisé sur le ou les champs de données.

Ces fonctions sont assez courtes et ne présentent pas de difficultés particulières du point de vue de la syntaxe à partir du moment où les notions de structure et de pointeurs sont bien maîtrisées. Cependant, il est parfois difficile de bien comprendre ce qu'elles font. Il est recommandé de bien les étudier et de vérifier leur comportement sur des exemples simples et sur les différents cas particuliers (selon les cas, liste vide, liste ne contenant qu'un seul élément, etc).

7.3.1 Création d'un élément et ajout d'un élément au début de la liste (en tête de liste)

Cette fonction prend en argument la valeur de l'entier à ajouter au début de la liste, ainsi que la liste (qui peut être `NULL`). Elle alloue la mémoire associée à un élément, initialise les différents champs et renvoie un pointeur sur le nouvel élément alloué par la fonction.

```
1  cellule_t *cons(int val, cellule_t *pListe){
2     cellule_t *el;
3
4     el =  malloc(sizeof(cellule_t));
5     if (el == NULL) {
6        return NULL;
7     }
8     el->donnee=val;
9     el->suivant = pListe;
10    return el;
11 }
```

Voici des exemples d'appel de cette fonction :

```
1 int main(){
2    cellule_t *ns;
3
4    ns=cons(3,ns);
5
6    ns=cons(2,ns);
7    ns=cons(1,ns);
8    return 0;
9 }
```

Vous pouvez télécharger une évolution détaillée de la mémoire avec le fichier Cons.pdf.

7.3.2 Parcours simple d'une liste

Parcours simple d'une liste : affichage

Pour afficher tous les éléments d'une liste il faut parcourir cette liste. Comme nous ne connaissons pas a *a priori* le nombre d'éléments de la liste, il n'est pas possible d'utiliser comme pour les tableaux une boucle **for** avec un compteur. Nous utiliserons donc une boucle **while** qui s'arrêtera lorsque nous n'aurons plus d'élément à afficher. Voici le code de la fonction d'affichage.

```
1 void Afficher_liste_int(cellule_t *liste){
2    cellule_t *cell=liste;
3    while (cell != NULL) {
4       fprintf(stderr, "%d ",cell->donnee);
5       cell = cell->suivant;
6    }
7    fprintf(stderr, "\n");
8 }
```

Nous mettons donc le pointeur vers le premier élément de la liste dans une variable locale nommée `cell` (ligne 2). L'instruction :

```
cell = cell->suivant;
```

permet de passer ("sauter") à l'élément suivant. La boucle s'arrête quand `cell` est `NULL`, c'est-à-dire qu'on n'a plus d'élément de liste à afficher.

Remarque : dans cette fonction, nous utilisons la fonction `fprintf` avec comme premier argument `stderr` et non `printf` comme habituellement. Comme `printf`, cette fonction affiche des messages dans le terminal (représenté ici par `stderr`). Mais la fonction `printf` *bufferise* les messages à afficher, c'est-à-dire les stocke dans un *buffer* et ne les écrit que quand le *buffer* est plein. Par conséquent, en cas de *segmentation fault* (erreur de segmentation à l'exécution), `printf` peut ne pas avoir encore affiché tous les

messages du programme (le buffer n'est pas plein) et on peut alors croire que le programme s'est arrêté plus tôt qu'il ne s'est arrêté en réalité. La fonction `fprintf` avec comme premier argument `stderr` ne *bufferise* pas les messages et tous ses appels donnent lieu à un affichage immédiat. En résumé, il faut utiliser `fprintf` + `stderr` pour les messages pour *debugguer* le programme et `printf` pour les autres messages.

Vous pouvez suivre l'évolution de l'état de la mémoire dans le fichier Afficher.pdf.

Parcours simple d'une liste : somme des éléments d'une liste

Voici le code d'une fonction qui calcule et retourne la somme des éléments d'une liste d'entiers.

```
1 int somme_list(cellule_t *ns){
2   cellule_t *cell = ns;
3   int r=0;
4   while (cell != NULL) {
5     r = r + cell->donnee ;
6     cell = cell->suivant;
7   }
8   return r;
9 }
```

La parcours est similaire à celui la fonction d'affichage `Afficher_liste_int` : on commence au début de la liste (le premier élément) et on s'arrête lorsqu'on n'a plus d'élément (`cell` contient `NULL`).

Vous pouvez télécharger une évolution détaillée de la mémoire avec le fichier somme_list.pdf.

Parcours simple d'une liste : retourner les éléments au carré

Voici le code d'une fonction qui calcule et retourne une liste dont les éléments sont les éléments au carré de la liste initiale.

```
1 cellule_t* liste_carres(cellule_t *ns){
2   cellule_t* newListe=NULL, *cell=ns;
3   while (cell!=NULL) {
4     newListe=cons(cell->donnee*cell->donnee, newListe);
5     cell=cell->suivant;
6   }
7   return newListe;
8 }
```

Le parcours est toujours le même mais cette fois il faut en plus construire une nouvelle liste. Pour cela nous utilisons la fonction `cons` que nous avons

écrite pour créer une liste, avec comme valeur à ajouter les éléments de la liste `ns` au carré. Cependant, le résultat n'est pas exactement celui que nous attendons : la nouvelle liste est à l'envers (cf. le déroulement détaillé dans le fichier liste_carres_it.pdf). Pour que la liste soit à l'endroit, il est par exemple possible de la retourner avec la fonction `rev` présentée page **??** (même si algorithmiquement il serait plus efficace d'insérer directement dans le bon ordre).

7.3.3 Parcours pouvant être interrompu : recherche un élément dans une liste

Nous avons maintenant une valeur et nous voulons savoir si un élément de la liste contient cette valeur, et si oui, avoir l'adresse de cet élément. Si l'élément est absent, notre fonction retournera `NULL`. Pour cela, l'agorithme est :

Pour chaque élément de la liste

Si c'est l'élément cherché, on a fini, retourner l'élément

Sinon, on continue

Si à la fin on n'a pas trouvé, c'est qu'il n'y est pas, on retourne alors NULL

Et voici l'implémentation en C de cette fonction :

```
1  cellule_t* rechercher(cellule_t* liste, int val){
2     while(liste != NULL){
3        if (liste->donnee == val) {
4           return liste;
5        }
6        else { /* else non obligatoire*/
7           liste=liste->suivant;
8        }
9     }
10    return NULL;
11 }
```

Vous pouvez suivre l'évolution détaillée de la mémoire dans le fichier Rechercher.pdf.

À la différence des fonctions précédentes, pour tester cette fonction, il faut envisager plusieurs cas. Il faut vérifier si elle fonctionne lorsque :

— l'élément à trouver est en 1er ?
— l'élément à trouver est en dernier ?
— l'élément à trouver est au milieu ?
— l'élément à trouver est absent ?

Un bon `main` serait donc :

```c
int main(){
  cellule_t * ns, *el;
  ns=cons(3,ns);
  ns=cons(2,ns);
  ns=cons(1,ns);
  el=rechercher(ns, 1);
  if(el==NULL || el-> donnee != 1)
    fprintf(stderr, "Erreur a la recherche du premier
       element\n");
  el=rechercher(ns, 2);
  if(el==NULL || el-> donnee != 2)
    fprintf(stderr, "Erreur a la recherche d'un element
        au milieu\n");
  el=rechercher(ns, 3);
  if(el==NULL || el-> donnee != 3)
    fprintf(stderr, "Erreur a la recherche d'un element
        a la fin\n");
  el=rechercher(ns, 4);
  if(el==NULL )
    fprintf(stderr, "Erreur a la recherche d'un element
        absent\n");
  return 0;
}
```

7.3.4 Parcours nécessitant de conserver l'élément précédent

Libérer une liste

L'objectif de cette fonction est de libérer la mémoire allouée pour la liste (cf. figure **??**)

Pour libérer une liste, l'algorithme assez intuitif est :

> Pour chaque élément de la liste
>
> > Libérer l'élément (avec `free`)
> >
> > Passer au suivant.

Il y a cependant un problème qui est que quand l'élément est libéré, on n'a plus accès à son champ suivant et on a donc perdu la suite de la liste... Il faut donc conserver l'adresse contenue dans le champ suivant dans une autre variable. L'implémentation est alors :

```c
cellule_t *Liberer_liste(cellule_t *liste){
   cellule_t *tmp;
   while(liste != NULL) {
      tmp = liste->suivant;
```

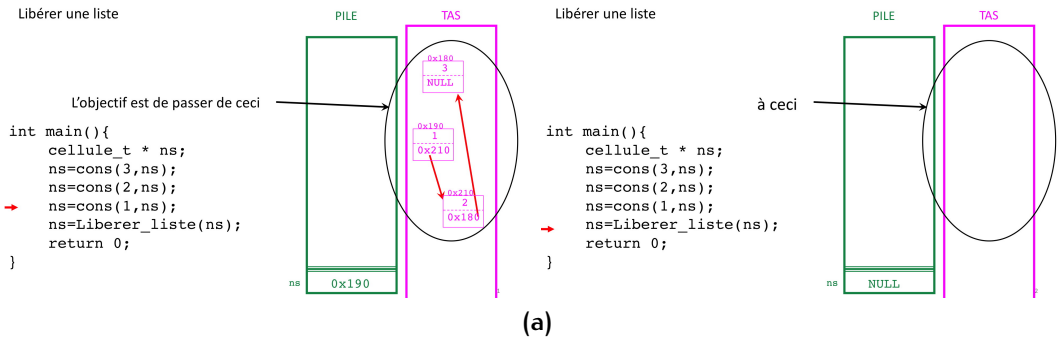


FIGURE 2 – Libération d'une liste allouée dans le tas. (a) Avant la libération. (b) Après la libération

```
5         free(liste);
6         liste = tmp;
7
8     }
9     return NULL;
10 }
```

Cette fonction retournera toujours NULL. Cela est pratique pour ne pas oublier de mettre à `NULL` le pointeur de tête de liste de la fonction appelante.

```
1  int main(){
2    cellule_t *ns;
3    ns=cons(1,cons(2,cons(3,NULL)));
4    ns=Liberer_liste(ns);
5    return 0;
6  }
```

Vous pouvez suivre l'évolution de l'état de la pile et du tas dans le fichier Liberer.pdf.

Inverser l'ordre d'une liste

Voici le code d'une fonction qui inverse l'ordre des éléments d'une liste.

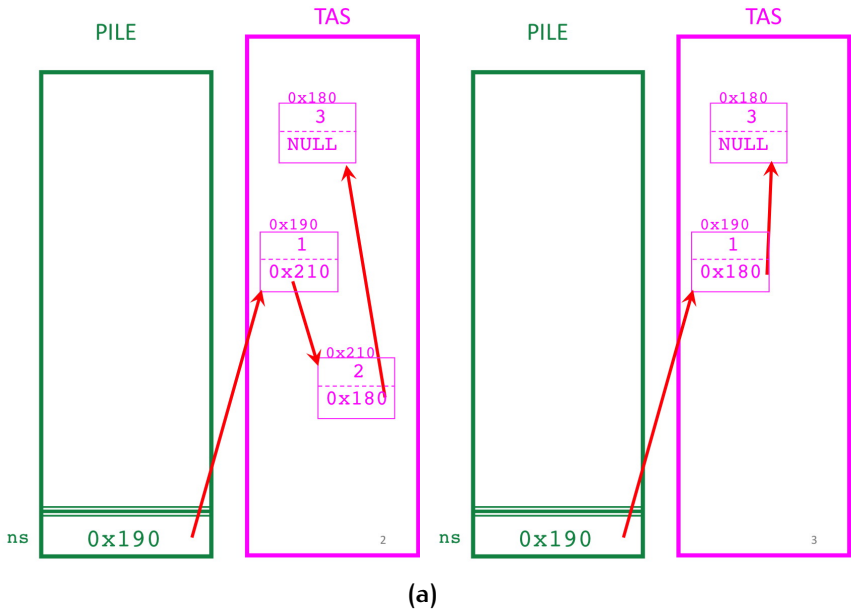
```
1  cellule_t* rev(cellule_t *ns){
2    cellule_t* newListe=NULL, *suiv, *cell=ns;
3    while (cell!=NULL) {
4      newListe=cons(cell->donnee, newListe);
5      suiv=cell->suivant;
6      free(cell);
7      cell=suiv;
8    }
9    return newListe;
```

```
10 }
```

Nous créons donc une nouvelle liste `newList` dans laquelle nous mettons d'abord le 1er élément, puis le second, puis le troisième, *etc.* Ainsi, ce qu'on a ajouté en premier dans cette nouvelle liste (le premier élément de de `ns`) se retrouve, à la fin de la fonction, en dernier de la liste `newList`. Nous avons choisi de libérer (désallouer, `free`) la liste `ns` en même temps que nous créons la liste `newList` pour éviter les fuites mémoire (la conservation dans le tas d'espaces mémoire alloués qui ne sont plus utilisés). Nous aurions pu aussi utiliser la fonction de libération de la liste mais cela est moins efficace algorithmiquement car cela nécessite de parcourir une deuxième fois la liste.

Supprimer un élément d'une liste

Nous souhaitons maintenant supprimer de la liste un élément dont nous avons la valeur. Par exemple, si nous supprimons l'élément 2 de la liste habituelle qui est en 2e position, il faut alors que le premier élément se mette à pointer vers le dernier élément (cf. figure **??**)



(a)

FIGURE 3 – Suppression du deuxième élément, de valeur 2

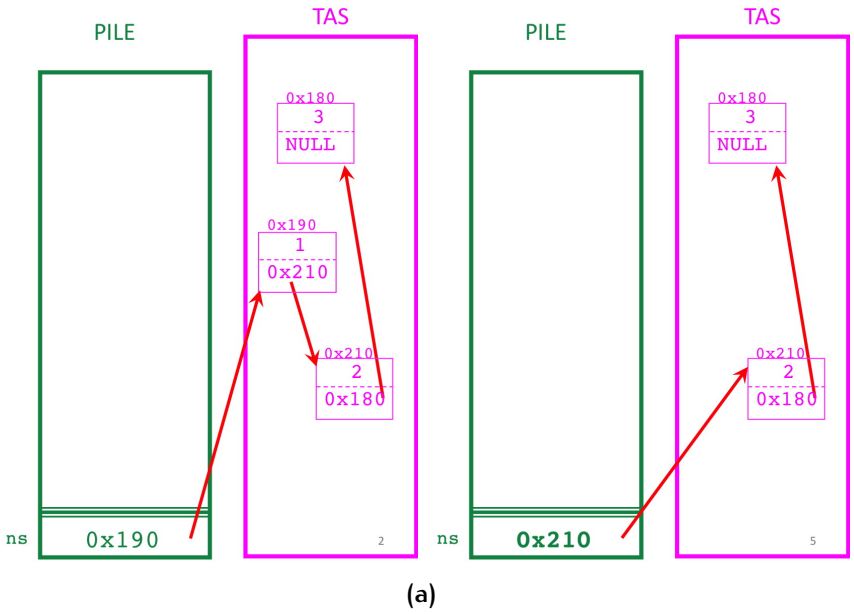
Les différentes étapes sont donc :

Trouver l'élément

Mettre à jour le chaînage

Libérer l'élément

Les cas est cependant différent lorsqu'on supprime le premier élément (cf. figure **??**. Il faut alors aussi **modifier** le pointeur qui est dans la fonction appelante (dans nos examples, le `main`). Il faudra donc retourner la tête de liste (le pointeur vers le premier élément), dont l'adresse pourra éventuellement changer.



(a)

FIGURE 4 – Suppression du premier élément, de valeur 1

L'algorithme pour une version itérative de cette fonction (la version récursive est plus simple, elle est présentée page **??**) est :

Trouver l'élément
Si on a bien trouvé l'élément
    S'il n'est pas le 1er
        On met à jour le chaînage
    Sinon
        On change la tête de liste
    On libère l'élément
    On retourne un pointeur vers le 1er élément

L'implémentation en C est :

```c
cellule_t *Supprimer_elt(cellule_t *liste, int val){
   cellule_t *tete, *precedent = NULL;
    /* On garde un pointeur sur la  tete */
   tete = liste;
   /* Trouver l'element*/
   while (liste != NULL && liste->donnee != val) {
      /* On garde un pointeur sur le precedent */
      precedent = liste;
      liste = liste->suivant;
   }
   if (liste) { /* Si j'ai trouve l'element */
      if (precedent) {/* S'il n'est pas le 1er */
         /* On met a jour le chainage*/
         precedent->suivant = liste->suivant;
      }
      else { /* S'il est le 1er */
         /* On change la tete de liste */
         tete = liste->suivant;
      }
      /* On libere l'element */
      free(liste);
   }
   /* On retourne un pointeur vers le 1er element */
   return tete;
}
```

Vous pouvez suivre l'évolution de la mémoire dans le fichier Supprimer.pdf
NB : le pointeur `NULL` correspondant à l'adresse 0, les conditions `(liste)` et `(liste != NULL)` sont équivalentes.

7.3.5 Parcours s'arrêtant au dernier élément

Concaténer deux listes

Concaténer deux listes revient à accrocher la seconde liste à la fin de la première. Il faut donc parcourir la première liste jusqu'à atteindre son dernier élément, puis lui accrocher la deuxième liste. La condition de continuation du **`while`** est donc différente cette fois : il faut s'arrêter lorsqu'on est au dernier élément, c'est-à-dire celui dont le champ `suivant` est `NULL`, et il faut donc continuer tant que ce n'est pas le cas.

Voici le code en C :

```
1  cellule_t *Concatener_it(cellule_t *liste1, cellule_t *
       liste2){
2  /* Renvoie la liste obtenue en accrochant liste2 a la
       fin de liste1 */
3      cellule_t *tmp;
4      if (liste1 == NULL) {
5          return liste2;
6      }
7      if (liste2 == NULL) {
8          return liste1;
9      }
10     tmp = liste1;
11     while (tmp->suivant != NULL) {
12         tmp = tmp->suivant;
13     }
14     tmp->suivant = liste2;
15     return liste1;
16 }
```

7.3.6 Autre parcours : parcourir deux listes (création d'une liste par fusion de deux listes ordonnées)

Et voici pour finir le code en C d'une fonction un peu complexe qui, à partir de deux listes ordonnées croissantes d'entiers, crée une nouvelle liste contenant tous les entiers des deux listes, en ordre croissant.

```
1  cellule_t *Fusion_it(cellule_t *l1, cellule_t *l2) {
2     cellule_t *liste_fusion=NULL, *next1=l1, *next2=l2;
3     int v;
4     while (next1 != NULL && next2 != NULL) {
5        if (next1->donnee < next2->donnee){
6           v = next1->donnee;
7           next1 = next1->suivant;
8        }
9        else if (next1->donnee == next2->donnee) {
10          v = next1->donnee;
11          next1 = next1->suivant;
12          next2 = next2->suivant;
13       }
14       else { // (next1->donnee > next2->donnee)
15          v = next2->donnee;
16          next2 = next2->suivant;
17       }
18       liste_fusion = cons(v, liste_fusion);
19    }
20    while (next1 != NULL) {
21       /* on a atteint la fin de l2, pas de l1 */
22       liste_fusion = cons(next1->donnee, liste_fusion);
23       next1 = next1->suivant;
24    }
25    while (next2 != NULL) {
26       /* on a atteint la fin de l1, pas de l2 */
27       liste_fusion = cons(next2->donnee, liste_fusion);
28       next2 = next2->suivant;
29    }
30    liste_fusion = rev(liste_fusion);
31    return liste_fusion;
32 }
```

8 LISTES CHAÎNÉES ET FONCTION RÉCURSIVES

8.1 INTRODUCTION ET PREMIÈRES FONCTIONS

La définition de la structure définie au chapitre précédent pour les listes chaînées a un caractère particulier : c'est une *définition récursive*. Par exemple dans

```
1 typedef struct _cellule_t cellule_t;
2 struct _cellule_t{
3   int donnee;
4   cellule_t *suivant;
5 };
```

une valeur de type `cellule_t` contient un pointeur vers une valeur de type `cellule_t`. Pour compléter l'ensemble des listes, on utilise le *pointeur nul* (constante symbolique `NULL`) pour désigner la liste vide.

Cela confère aux listes elles-mêmes une *structure récursive* : une liste est soit un *pointeur nul* soit composée d'un premier élément suivi... d'une liste. En résumé on peut complètement caractériser l'ensemble des structures de listes chaînées de la manière suivante :

> `cs` est une liste si et seulement si
> — `cs` est la liste vide `NULL`, ou
> — sinon, `cs` contient un premier élément dont la valeur est donnée par `cs->donnee` et la suite est donnée par `cs->suivant`

Cette vision récursive des listes permet naturellement de concevoir leur manupulation de manière récursive. Par exemple, pour afficher tous les éléments d'une liste, on met en application ce principe très simple :

1. Afficher le premier élément.
2. Afficher la suite de la liste.

Pour être complète, la définition de cet *algorithme récursif* « afficher la liste » doit considérer le cas de la liste vide :

> Pour « afficher la liste » `cs` :
> — Si `cs` est `NULL` , ne rien afficher (**cas de base**) ;
> — Sinon (**cas général)** :
> — afficher `cs->donnee;`
> — « afficher la liste » `cs->suivant`.

Le langage C autorise de telles *définitions récursives*

```
1  void Afficher_liste_int(cellule_t *cs) {
2    if (cs == NULL) {
3      printf("\n");
4    } else {
5      printf("%d ",cs->donnee);
6      Afficher_liste_int(cs->suivant);
7    }
8  }
```

Pour que l'affichage soit plus compact, on a ici choisi d'afficher tous les éléments sur une seule ligne et de passer à la ligne après le dernier élément. Ainsi, lorsque la liste est vide (`NULL`), on passe à la ligne (`printf("\n")`) et on sépare les éléments affichés par un espace (`printf("%d ",cs->donnee)`).

Ce principe s'applique aussi facilement pour les fonctions dont le résultat est obtenu en parcourant les structures de listes. Par exemple :

Pour renvoyer «la somme des éléments de la liste » `cs` :
- si `cs` est `NULL` alors renvoyer `0`;
- sinon, renvoyer l'addition de `cs->donnee` et « la somme des éléments de la liste » `cs->suivant`.

```
1 int somme_liste_int(cellule_t *cs) {
2   if (cs == NULL) {
3     return 0;
4   } else {
5     return (cs->donnee + somme_liste_int(cs->suivant));
6   }
7 }
```

Dans
`return (cs->donnee + somme_liste_int(cs->suivant))`
l'application `somme_liste_int(cs->suivant)` est appelée *appel récursif* de la fonction `somme_liste_int`.

8.2 EXÉCUTION D'UNE FONCTION RÉCURSIVE (RÉCURSION SUR LES ENTIERS)

Pour comprendre ce qui se passe lorsqu'une fonction récursive est appelée considérons un exemple de définition récursive d'une fonction sur les entiers : la fonction qui renvoie la somme des entiers de 0 à n. Pour avoir une expression simple de cette fonction, remarquons que la somme $0 + \cdots + n$ est égale à la somme $n + \cdots + 0$.

Voici alors un algorithme récursif pour renvoyer « la somme des entiers de 0 à n » :
— si $n = 0$ alors renvoyer 0 ;
— sinon, renvoyer l'addition de n et « la somme des entiers de 0 à $n-1$ ».

D'où la définition récursive

```
1 int somme_int(int n) {
2   if (n==0) {
3     return 0;
4   } else {
5     return (n + somme_int(n-1));
6   }
7 }
```

Pour expliquer le principe d'évaluation d'une telle fonction, on peut développer les appels récursifs.

Prenons l'exemple simple de l'application `somme_int(3)` :

```
(1) somme_int(3)  =  3 + somme_int(2)
(2)               =  3 + (2 + somme_int(1))
(3)               =  3 + (2 + (1 + somme_int(0)))
(4)               =  3 + (2 + (1 + 0))
(5)               =  3 + (2 + 1)
(6)               =  3 + 3
(7)               =  6
```

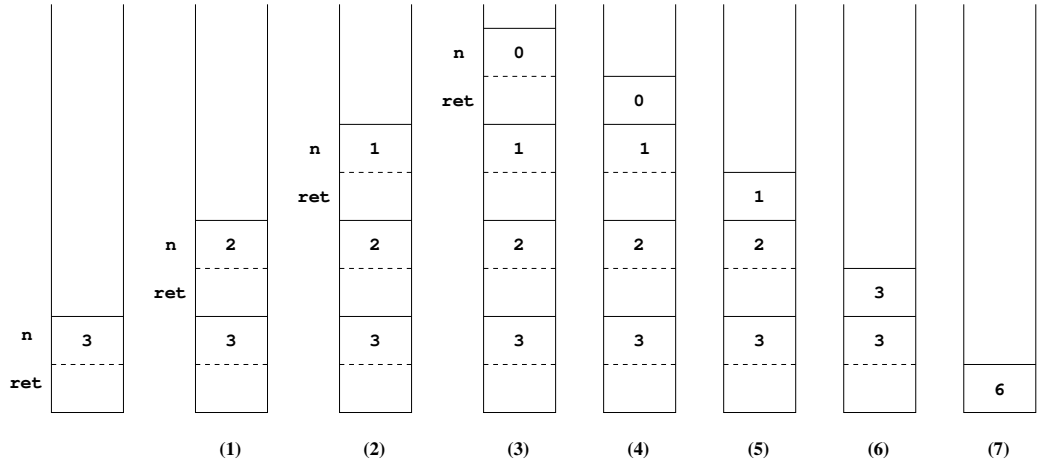
Ces 7 étapes d'évaluations se composent :
de (1) à (4) phase de « développement » des appels récursifs avec mise en attente des additions ;
de (5) à (7) calcul effectif des additions avec les résultats des appels récursifs.

Le « tournant » a lieu à l'étape (4) lorsque l'appel récursif avec 0 permet de mettre fin à la phase de développement puisqu'alors la valeur de retour de la fonction est immédiatement donnée (`return 0`). Cette valeur permet d'avoir celle de l'appel précédent (`somme_int(1)`=1+0=1) ; puis celle de l'appel précédent (`somme_int(2)`=2+1=3) et enfin, celle de l'appel initial (`somme_int(3)`=3+3=6).

Lorsque que le code de la fonction `somme_int` est exécuté, le « développement » des appels récursifs se traduit par un « empilement » de la mémoire réservée pour les appels des fonctions dans la pile et une « mise en attente » des fonctions en attendant les résultats des appels (cf. figure **??**).

Pour expliquer le mécanisme d'appel de fonction, nous avons montré comment les arguments des fonctions sont mis sur la pile pendant l'évaluation du corps de la fonction. Compliquons un peu ce modèle en ajoutant un autre élément sur la pile lors de l'appel d'une fonction : une place pour

recevoir le résultat. Appelons `ret` cette place. Avec cette nouvelle convention d'appel, la figure suivante illustre l'évolution de la pile lors des 7 étapes du calcul de `somme_int(3)`



8.3 SPÉCIFICATIONS ÉQUATIONNELLES

Il est d'usage en mathématique de définir les fonctions récursives (ou les suites récursives) de la manière suivante (ici, l'élévation de x à la puissance entière n) :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{sinon} \end{cases}$$

ou encore, de manière équivalente :

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x \times x^n \end{aligned}$$

Cette manière repose sur le fait que l'ensemble des entiers naturels peut être *construit* à partir de la constante 0 est de l'opération $n \mapsto n + 1$ (la fonction *successeur*).

Cette caractéristique existe aussi pour l'ensemble des listes que nous avons défini (rappel) :

> `cs` est une liste si et seulement si
> — `cs` est la liste vide `NULL`, ou
> — sinon, `cs` contient un premier élément dont la valeur est donnée par `cs->donnee` et la suite est donnée par `cs->suivant`.

La liste vide `NULL` correspond au 0 des entiers et la fonction `cons` (création de liste, **??**) peut correspondre à la fonction successeur des entiers. À la manière de ce qui se fait pour les entiers en mathématique, on peut *spécifier*

avec des équations des fonctions récursives sur les listes. Par exemple, on peut définir ainsi la (fonction de calcul de la) taille d'une liste :

$$\texttt{taille}(cs) = \begin{cases} 0 & \text{si } cs = \texttt{NULL} \\ 1 + \texttt{taille}(cs\texttt{->suivant}) & \text{sinon} \end{cases}$$

ou

$$\begin{aligned} \texttt{taille(NULL)} &= 0 \\ \texttt{taille(cons(x,cs))} &= 1 + \texttt{taille(cs)} \end{aligned}$$

La première forme de définition peut se transcrire assez directement comme une définition en C :

```
1 int taille(cellule_t *cs) {
2   if (cs == NULL) {
3     return 0;
4   } else {
5     return 1 + taille(cs->suivant);
6   }
7 }
```

La seconde est plutôt utile comme outil de *simulation* du calcul :

```
taille(cons(67, cons(34, cons(98, NULL))))
= 1+taille(cons(34, cons(98, NULL)))
= 1+(1+taille(cons(98,NULL)))
= 1+(1+(1+taille(NULL)))
= 1+(1+(1+0))
= 1+(1+1)
= 1+2
= 3
```

Pour donner un autre exemple, voici comment on peut spécifier la fonction `rechercher`

$$\texttt{rechercher}(cs, v) = \begin{cases} \texttt{NULL} & \text{si } cs = \texttt{NULL} \\ cs & \text{sinon, si } cs\texttt{->donnee} = v \\ \texttt{rechercher}(cs\texttt{->suivant}, v) & \text{sinon} \end{cases}$$

Ce qui donne la définition C suivante :

```
1 cellule_t *rechercher(cellule_t cs, int v) {
2   if (cs == NULL) {
3     return NULL;
4   } else {
5     if (cs->donnee == v) {
6       return cs;
7     } else {
8       return rechercher(cs->suivant, v);
```

```
9       }
10    }
11  }
```

Voici maintenant une spécification de la fonction de `cellule_t *Supprimer_elt_rec(cellule_t *liste, int val);` dont l'algorithme récursif est plus simple que l'algorithme itératif (présenté dans la partie **??** page **??**).

$$\texttt{Suppr}(cs,v) = \begin{cases} \texttt{NULL} & \text{si } cs = \texttt{NULL} \\ \texttt{cs->suivant} & \text{sinon, si } cs\texttt{->donnee} == v \\ \texttt{cons (cs->donnee, Suppr(cs->suivant,v))} & \text{sinon} \end{cases}$$

Ce qui donne la définition C suivante :

```
1  cellule_t * Supprimer_el_rec(cellule_t *liste, int val)
     {
2    cellule_t * tmp;
3    if (liste){
4       if(liste->donnee == val){
5          tmp = liste->suivant;
6          free(liste);
7          liste = tmp;
8       }
9       else {
10         liste->suivant =
11            Supprimer_el_rec(liste->suivant, val);
12      }
13   }
14   return liste;
15 }
```

9 FILES D'ATTENTE

Les structures chaînées que nous avons utilisées pour les listes (chapitre **??**) permettent de définir facilement les opérations d'ajout et de retrait en tête de la structure. Il existe une autre manière assez standard d'ajouter et de retirer des éléments dans une structure chaînée :

— ajouter en fin de liste ;
— retirer en tête de liste.

Cette manière d'utiliser les listes est connue sous le nom de *file d'attente* car elle modélise effectivement les files d'attentes à un guichet, par exemple, où le premier arrivé sera le premier servi [1].

Pour réaliser les structures de files d'attente, nous utiliserons les mêmes structures de cellules que pour les listes :

```
1 typedef struct _cellule_t cellule_t;
2 struct _cellule_t{
3   int donnee;
4   cellule_t *suivant;
5 };
```

Ce qui va changer, c'est l'information que nous garderons sur ces structures chaînées.

9.1 LA CHANDELLE PAR LES DEUX BOUTS

Avec les listes, les opérations d'ajout et de retrait en tête sont simples car quand on a une valeur qui désigne une liste, on a une valeur de type `cellule_t*` : un pointeur vers la *tête* de la liste. Un pointeur vers la tête de liste suffira donc pour réaliser l'opération de retrait des files d'attente. En revanche, cette seule information est très inefficace pour l'opération d'ajout *en fin* de liste. En effet, pour ajouter en fin de liste, il faut alors parcourir toute la liste jusqu'à son dernier élément puis ajouter le nouvel élément après celui-ci (voir exercice 30, question 2). L'opération d'ajout devient alors de plus en plus coûteuse au fur et à mesure que la taille de la file grandit.

1. En anglias, on parle de *FIFO : Fisrt In First Out*.

9.1.1 Le type `fifo`

Pour avoir une opération d'ajout en fin de liste à « coût constant », quelle que soit la taille de la liste, il suffit de mémoriser le pointeur sur le dernier élément de la liste. Comme nous voulons aussi garder le pointeur sur le premier élément pour faciliter l'opération de suppression, nous définissons une structure à deux éléments :

```
1  struct s_fifo {
2    cellule_t* first;
3    cellule_t* last;
4  };
5  typedef struct s_fifo fifo;
```

Une file d'attente est une valeur de type `fifo`. Son contenu est concrètement celui d'une structure chaînée de cellules (type `cellule_t`) pour laquelle nous conserverons un pointeur vers le premier élément (champ `first`) et un pointeur vers le dernier (champ `last`). Comme pour les listes chaînées, nous pourrons avoir accès à la totalité des valeurs contenues dans la file à partir de la valeur (qui est une adresse) donnée par le champ `first`.

9.1.2 La file d'attente vide

Pour créer une file d'attente, il suffit de créer une structure de type `fifo`. À sa création, une file d'attente est *vide*. Comme la suite chaînée de cellules qui donne son contenu est vide, on initialise ses deux champs avec la valeur `NULL`. La fonction de création d'une file d'attente ne dépend d'aucune valeur particulière : c'est une fonction *sans argument* ; plus exactement, avec une suite d'arguments vide.

```
1 fifo new_fifo() {
2   fifo r;
3   r.first = NULL;
4   r.last = NULL;
5   return r;
6 }
```

Remarque : Nous n'avions pas eu besoin, concernant les listes de définir une fonction de création de liste. Le pointeur « `NULL` » suffisait à remplir ce rôle. Par exemple, on déclarait simplement une liste initialement vide en écrivant : `cellule_t* liste = NULL;`

La situation est maintenant différente pour les files d'attente vides. On déclarera (et initialisera) une liste d'attente avec

```
1   cellule_t file = new_fifo();
```

Lorsque nous avons eu à définir des fonctions sur les listes il fallait, en général, traiter le cas de la liste vide. On a fait cela avec le simple test `if (... == NULL)`. Comme nous avons choisi de représenter les files d'attente par une structure encapsulant deux pointeurs, ce simple test ne suffit plus. C'est pourquoi on définit une fonction dédiée au test de la vacuité d'une file d'attente :

```
1 int empty_fifo(fifo xs) {
2   return (xs.first == NULL && xs.last == NULL);
3 }
```

Cette définition est *cohérente* avec la définition de la fonction `new_fifo` qui renvoie une file d'attente initialement vide, c'est-à-dire, une structure dont les deux champs ont la valeur `NULL`.

IL EST IMPORTANT de noter ici que nous aurions pu faire d'autres choix. Par exemple, puisque c'est le champ `first` des strutures `fifo` qui donne accès au contenu de la file d'attente, nous aurions pu nous contenter de

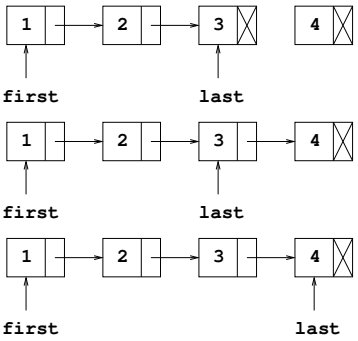
```
1 int empty_fifo(fifo xs) {
2   return (xs.first == NULL);
3 }
```

Nous reviendrons sur ce point.

9.2 AJOUT

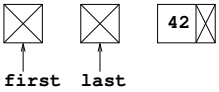
L'ajout d'une élément dans une file d'attente se fait par création d'une nouvelle cellule que l'on raccorde au dernier élément de la file. L'élément ajouté devient le dernier élément de la file d'attente. Cette opération est illustrée par les trois étapes suivantes :

1. créer une nouvelle cellule (contenant 4)
2. chaîner la nouvelle cellule
3. mettre à jour le champ `last`



Cette opération suppose toutefois que le dernier élement existe, c'est-à-dire que la file n'est pas vide. L'opération d'ajout dans une file d'attente nécessite donc de faire attention à ce *cas particulier* : l'ajout dans une file vide. Dans ce cas, les choses se passent ainsi :

1. créer une nouvelle cellule



2. initialiser les champs `first` et `last`

Notez comment, qu'après l'ajout d'un premier élément à une file vide, les champs `first` et `last` pointent vers la même cellule, c'est-à-dire contiennent la même valeur (adresse).

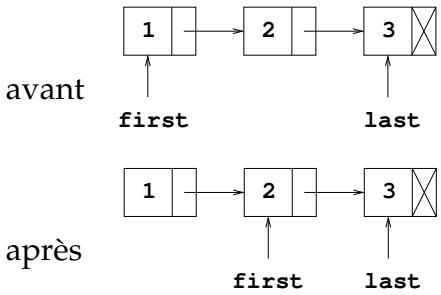
Ces précautions prises, nous pouvons définir la fonction `add` d'ajout d'un élément `x` à une file d'attente `xs`. Sa valeur de retour est la nouvelle file d'attente.

```
1  fifo add(int x, fifo xs) {
2    /* creation de la nouvelle cellule */
3    cellule_t *c = malloc(sizeof(cellule_t));
4    c->donnee = x;
5    c->suivant = NULL;
6    if (empty_fifo(xs)) {
7      /* le cas particulier */
8      xs.first = c;
9      xs.last = c;
10   } else {
11     /* le cas general */
12     xs.last->suivant = c;
13     xs.last = c;
14   }
15   return xs;
16 }
```

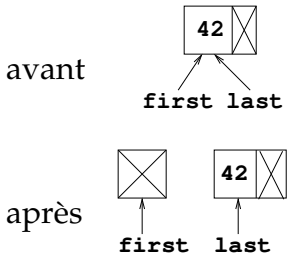
Remarquez comment, dans le cas général (lignes 14 à 14), on a chaîné le dernier élément de la file d'attente pointé par `last` avec la cellule `c` nouvellement créée.
Attention, xs n'est pas un pointeur; `xs.last` est donc bien l'instruction donnant accès au champ `last` de la structure de type `fifo xs`. Par contre, le champs `last` est lui de type `cellule_t*` (un *pointeur* vers une structure `cellule_t`). Par conséquent, pour accéder au champ `suivant` de la structure pointée par `last`, on utilise l'opérateur `->`, d'où `xs.last->suivant`.

9.2.1 Retrait

L'opération de retrait d'une valeur en tête de liste est quasi immédiate : il suffit de modifer la valeur du champ `first` :



Il y a cependant un problème si l'on applique le même principe sur une file ne contenant que 1 seul élément :

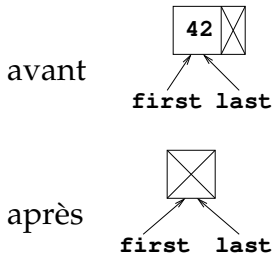


Le champs `last` n'a pas été modifié et pointe toujours sur l'élément à supprimer.

Ce qui n'est pas cohérent avec la manière dont la fonction `new_fifo` crée une file d'attente vide. Dans le cas du retrait de l'ultime élément d'une file d'attente, il faut également mettre à jour le champ `last`. Le cas de la liste à un élément est donc un cas particulier à traiter à part.

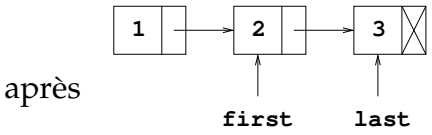
```
1  fifo pop(fifo xs) {
2    if (!empty_fifo(xs)) {
3      xs.first = xs.first->suivant;
4      if (xs.first == NULL) {
5        /* Le cas particulier */
6        xs.last = NULL;
7      }
8    }
9    return xs;
10 }
```

Avec notre traitement du cas particulier (test `(xs.first == NULL)`) on obtient :



Remarque : La fonction `pop` est *a priori* partielle : elle n'a pas de sens si la file d'attente est vide. Toutefois, on peut « compléter » assez naturellement la fonction en décidant que `pop(NULL)` est égal à `NULL`. C'est ce qui est fait dans la définition ci-dessus : le test `(!empty_fifo(xs))` laisse la file d'attente vide inchangée.

GESTION MÉMOIRE Notre fonction de retrait est minimale : elle se contente de rendre le premier élément inaccessible mais il existe encore en mémoire comme l'illustre la figure

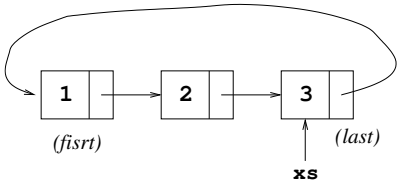


Ce choix de conception peut faire courrir le risque d'une *fuite mémoire*. On préférera donc donner une version de `pop` qui supprime *effectivement* le premier élément. Dans ce cas, on sauvegarde la valeur du pointeur `xs.first` avant sa modification, puis on libère la mémoire après la modification.

```
1  fifo pop(fifo xs) {
2    if (!empty_fifo(xs)) {
3      cellule_t* tmp = xs.first;
4      xs.first = xs.first->suivant;
5      if (xs.first == NULL) {
6        /* Le cas particulier */
7        xs.last = NULL;
8      }
9      free(tmp);
10   }
11   return xs;
12 }
```

9.3 LISTES CIRCULAIRES

Il est possible définir autrement une file en utilisant des listes dites **circulaires** où l'on conserve un pointeur non plus sur le premier mais sur le dernier élément et où le dernier élément pointe sur le premier. On a alors facilement accès au dernier mais aussi au premier élément car on peut utiliser le champ `suivant` du dernier élément pour savoir où commence la liste, comme l'illustre la figure ci-dessous :



Cette structure de *liste circulaire* va nous permettre de réaliser les fonctions d'ajout et de retrait dans les files d'attente sans avoir besoin d'une structure à deux pointeurs.

9.3.1 Un nouveau type `fifo`

En fait, pour définir une liste circulaire, notre structure `cellule_t` suffit :

```
1 struct _cellule_t {
2   int donnee;
3   struct _cellule_t* suivant;
4 };
5 typedef struct _cellule_t* fifo;
```

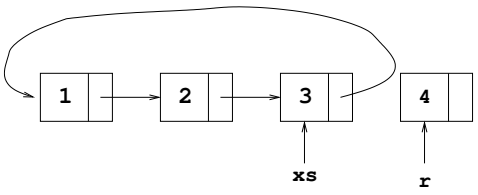
Avec cette définition, du point de vue des structures de données concrètes en C, les listes simples du chapitre **??** et les files d'attente sont identiques. Du coup, cela résoud le problème que nous avions avec la file d'attente vide. Avec cette nouvelle définition, il n'y en a plus qu'une : le pointeur `NULL`.

La différence entre les listes simples et les files d'attente tiendra donc au caractère circulaire de l'usage des listes chaînées pour les files d'attente ainsi qu'à la manière dont nous définissons l'ajout et le retrait dans les structures.

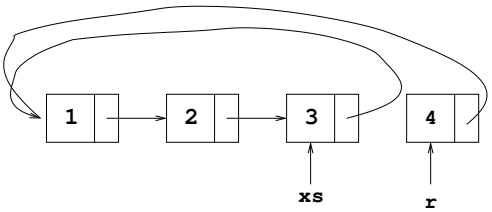
9.3.2 Ajout en tête

Soit `xs` une file d'attente, c'est-à-dire, un pointeur vers un élément d'une liste chaînée circulaire. Ajouter un élément à `xs` consiste à ajouter une cellule « après » `xs` qui est le dernier élément et chaîner ce nouvel élément avec celui qui suivait `xs` (avant l'ajout). Il faut donc réaliser les trois étapes suivantes :

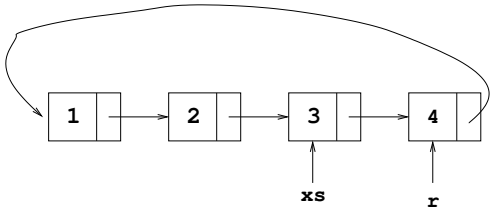
1. créer une nouvelle cellule `r`



2. la relier au premier



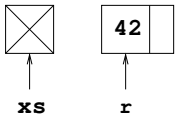
3. chaîner `r` dans la file



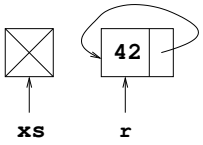
Le début de la file d'attente résultant de l'ajout en tête est donnée par `r`.

Naturellement ces trois étapes ne sont réalisables que si la file d'attente n'est pas vide : on aurait alors du mal à chaîner quoique se soit « après » le pointeur `NULL`! On retrouve donc avec la représentation des files d'attente par des listes circulaires le cas particulier que nous avions déjà rencontré : l'ajout dans une file vide. Dans ce cas, les opérations à réaliser sont :

1. créer une nouvelle cellule `r`



2. la chaîner à elle-même



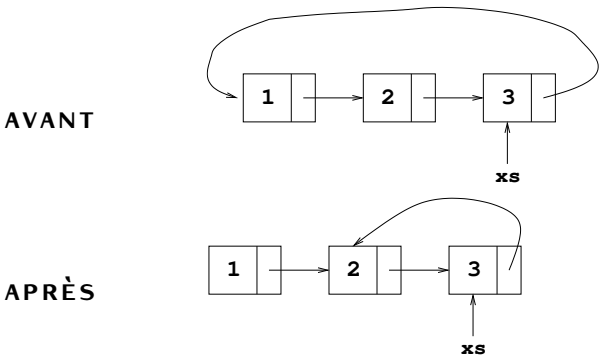
Et ici encore, le début de la file d'attente résultant de l'ajout en tête est donnée par `r`.

On définit la fonction d'ajout de cette manière :

```
1  fifo add(int x, fifo xs) {
2    fifo r = malloc(sizeof(struct _cellule_t));
3    r->donnee = x;
4    if (xs == NULL) {
5      /* le cas particulier:
6         on chaine r sur elle meme */
7      r->suivant = r;
8    }
9    else { /* cas general*/
10     r->suivant = xs->suivant;
11     xs->suivant = r;
12   }
13   return r;
14 }
```

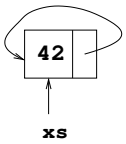
9.3.3 Retrait

L'opération de retrait du premier élément, dans le cas des listes ciculaires, est similaire à celle utilisée pour la structure à deux pointeurs : faire pointer le « premier élément » de la file d'attente vers la cellule qui le suit. On aura :



La différence tient simplement à la manière dont on accède au « premier élément ». Si `xs` est notre liste circulaire, son premier élément est celui pointé par `xs->suivant`. La suppression de celui fait que la nouvelle tête de liste sera `xs->suivant->suivant`. Après l'opération de retrait, `xs` n'a pas été modifié mais son champ `suivant` l'a été.

Attention encore ici au cas particulier de la file d'attente qui ne contient qu'un seul élément (voir figure ci-dessous) :



On peut identifier ce cas car alors la valuer de `xs` et de `xs->suivant` sont identiques. Et, dans ce cas, le résultat doit être la file vide, c'est-à-dire, le pointeur `NULL`.

Nous considérons encore ici que le retrait d'une file d'attente vide laisse cette file vide, sans provoquer d'erreur et libère la mémoire allouée pour l'élément supprimé.

```
1  fifo pop(fifo xs) {
2    cellule_t *tmp;
3    if (xs == NULL) {
4      /* File d'attente vide */
5      return NULL;
6    }
7    else if (xs->suivant == xs) {
8      /* Le cas particulier */
9      free(xs);
10     return NULL;
11   }
12   else {
13     tmp=xs->suivant;
14     xs->suivant = xs->suivant->suivant;
15     free(tmp);
16     return xs;
17   }
```

```
18 }
```

GESTION MÉMOIRE Pour éviter les risques de fuite mémoire, il faut libérer la cellule occupée par l'élément supprimé. C'est-à-dire `xs->suivant`, ou `xs` lui-même dans le cas particulier d'une file à un seul élément.

```
1  fifo pop(fifo xs) {
2    if (xs == NULL) {
3      /* File d'attente vide */
4      return NULL;
5    }
6    else if (xs->suivant == xs) {
7      /* Le cas particulier */
8      free(xs);
9      return NULL;
10   } else {
11     fifo tmp = xs->suivant;
12     xs->suivant = xs->suivant->suivant;
13     free(tmp);
14     return xs;
15   }
16 }
```

9.3.4 Parcours d'une liste circulaire

Aucun des champs `suivant` d'une liste circulaire (non vide) ne contient la valeur `NULL`. On ne peut donc pas parcourir une liste circulaire comme nous le faisions pour les listes simples avec une boucle de la forme

```
1    while (xs != NULL) {
2      ...
3      xs = xs->suivant;
4    }
```

Toutefois, dans une liste circulaire `xs` (non vide), nous connaissons son dernier élément (celui pointé par `xs` lui-même) et son premier élément (celui pointé par `xs->suivant`). On peut donc partir du premier élément, passer à l'élément « suivant », etc, tant que l'on est pas arrivé à `xs`. Voici à titre d'exemple une fonction d'affichage pour les files d'attente :

```
1  void affiche_fifo(fifo xs) {
2    printf("[");
3      if (xs != NULL) {
4        // On part premier element
5        fifo p = xs->suivant;
```

```
6         while(p != xs) {
7           printf("(%d)",p->donnee);
8           p = p->suivant;
9         }
10        printf("(%d)",p->donnee);
11      }
12      printf("]\n");
13  }
```

Notez que

— on ne boucle que si la liste est non vide (test `(xs != NULL)`);

— en sortie de boucle, il ne faut pas oublier d'afficher la valeur contenue dans le dernier élément.