Structures de données (LU2IN006)

Nawal Benabbou

Licence Informatique - Sorbonne Université

2021-2022

Matrice

Un premier type de données abstrait : les matrices

Les matrices permettent d'organiser les données "en tableau". Plus précisément, une matrice est un ensemble de cases, repérées par un index à une ou deux dimensions (parfois plus). Chaque case est représentée de manière unique par son index.

Implémentation en langage C

Matrice à une dimension :

- Les matrices à une dimension (ligne ou colonne) correspondent exactement aux tableaux C à une dimension (voir cours précédent).

Matrice à deux dimensions :

- Il est possible de coder une matrice à deux dimensions (notée `m`) par un tableau C à une dimension (noté `t`) en "aplatissant la matrice". Il suffit de faire correspondre la case `m[i][j]` avec la case `t[i*n+j]` où n est le nombre de colonnes de la matrice.
- Une matrice peut être codée par un tableau C à deux dimensions. Comme il peut être vu comme un tableau de tableaux, il est représenté par un pointeur sur pointeur en langage C.

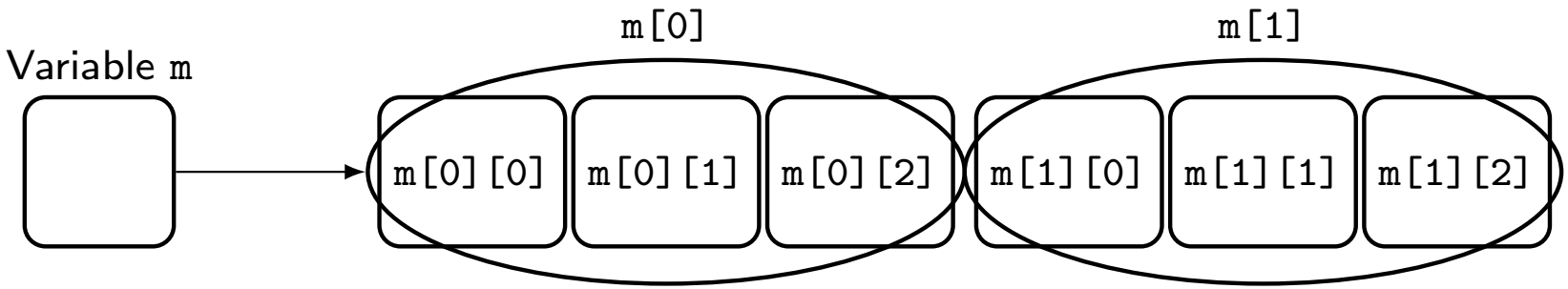
Les tableaux à deux dimensions en langage C

Allocation statique

En mémoire, les données sont stockées les unes après les autres, ligne par ligne. Un tableau C à deux dimensions peut être alloué de différentes manières. Par exemple :

- int m[2][3] ; permet d'allouer l'espace mémoire pour un tableau 2D de taille 2×3 contenant des entiers, sans initialiser ses valeurs.
- int m[2][3] = $\{\{1,5,4\}, \{3,2,1\}\}$; permet de faire les deux.
- int m[2][3] = $\{1,5,4,3,2,1\}$; permet de faire la même chose.

Remarque : `m` est ici de type `int**`.



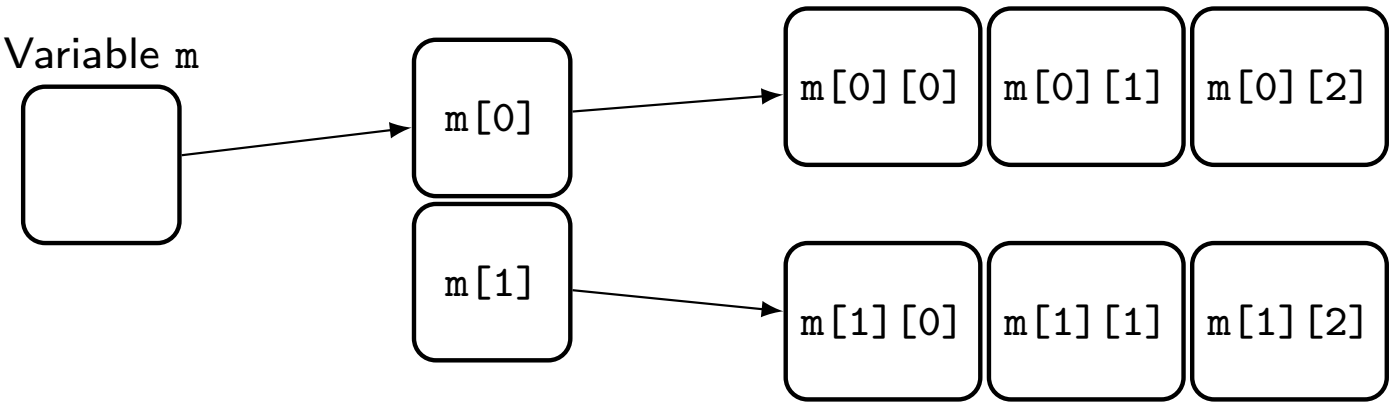
Les tableaux à deux dimensions en langage C

Quand la mémoire ne possède pas de zone espace contiguë assez grande, on préfère réaliser une allocation dynamique plutôt que statique.

Allocation dynamique

Allouer dynamiquement un tableau à une dimension, dont chaque case pointe sur un tableau à une dimension alloué dynamiquement. Exemple avec le même tableau à deux dimensions :

```
1  int i;
2  int** m = (int**) malloc(2*sizeof(int*));
3  for (i=0;i<2;i++){
4     m[i] = (int*) malloc(3*sizeof(int));
5  }
```



Les tableaux à deux dimensions en langage C

Désallocation : libérer les sous-tableaux, avant de libérer le tableau principal.

```
1  int i;
2  for (i=0;i<2;i++){
3      free(m[i]);
4  }
5  free(m);
```

Gestion des problèmes d'allocation : libérer la mémoire allouée avant l'erreur.

```
1  int i,j;
2  int** m = (int**) malloc(2*sizeof(int*));
3  if (m == NULL){
4      return NULL;
5  }
6  for (i=0;i<2;i++){
7      m[i] = (int*) malloc(3*sizeof(int));
8      if (m[i]==NULL){
9          for (j=0;j<i;j++){
10             free(m[j]);
11         }
12         free(m);
13         return NULL;
14     }
15 }
```

Complexité des opérations sur les données

Pour une matrice avec un index à une dimension (tableau).

Recherche d'un élément

On recherche si l'élément est dans le tableau, et on retourne l'indice si l'élément y est. L'opération de recherche est en $O(n)$. On verra que l'on peut obtenir une meilleure complexité si le tableau est trié.

Accès à un élément

Si on connaît l'indice de l'élément, l'opération est en $O(1)$ (temps constant).

Suppression d'un élément

Si on connaît l'indice de l'élément à supprimer, on peut décider de mettre la case à une valeur particulière (par exemple "-1") pour indiquer qu'elle est "vide". Dans ce cas, l'opération est en $O(1)$. Si par contre on souhaite décaler les valeurs pour laisser les cases contiguës, alors l'opération devient en $O(n)$.

Insertion d'un élément

On insère dans la première case libre (s'il y en a une). L'opération est en $O(1)$ si les cases sont maintenues contiguës, et en $O(n)$ sinon. Par contre, si on dépasse la taille du tableau, cette structure de données n'est peut-être pas adaptée.

Liste

Un autre type de donnée abstrait : les listes

Une liste est définie par une suite finie d'éléments $(x_1, \ldots, x_n)$. La liste est vide si $n = 0$. On passe de la case contenant l'élément x_i à la case contenant l'élément x_{i+1} par une fonction *successeur*. Une liste (non vide) a deux extrémités, que l'on appelle *début* et *fin*.

Implémentation en langage C

Une liste peut être codée par un tableau C à une dimension, mais en l'absence d'indices ou de tri sur les éléments, on n'a aucun intérêt à utiliser de la mémoire contiguë, qui consomme beaucoup de place en mémoire et qui pose des problèmes lors de l'insertion et de la suppression d'un élément. On privilégie une implémentation par liste chaînée.

Liste chaînée

Liste (simplement/doublement) chaînée

Une liste chaînée est définie par une chaîne linéaire, autrement dit par un ensemble de cellules possédant chacune un élément qui pointe sur un autre élément de la liste de sorte à former une "ligne". La liste est dite *simplement chaînée* si elle ne possède qu'une fonction *successeur*, et *doublement chaînée* si elle possède en plus une fonction *prédécesseur* permettant d'accéder à l'élément précédent.

Implémentation avec `struct`

En langage C, on va coder une cellule à l'aide d'un `struct`. Exemple pour une liste composée de paires (lettre, nombre), on peut utiliser le `struct` suivant :

```
1  typedef struct cellule {
2      char lettre;
3      int nombre;
4      struct cellule *suiv;
5
6  } Cellule;
```

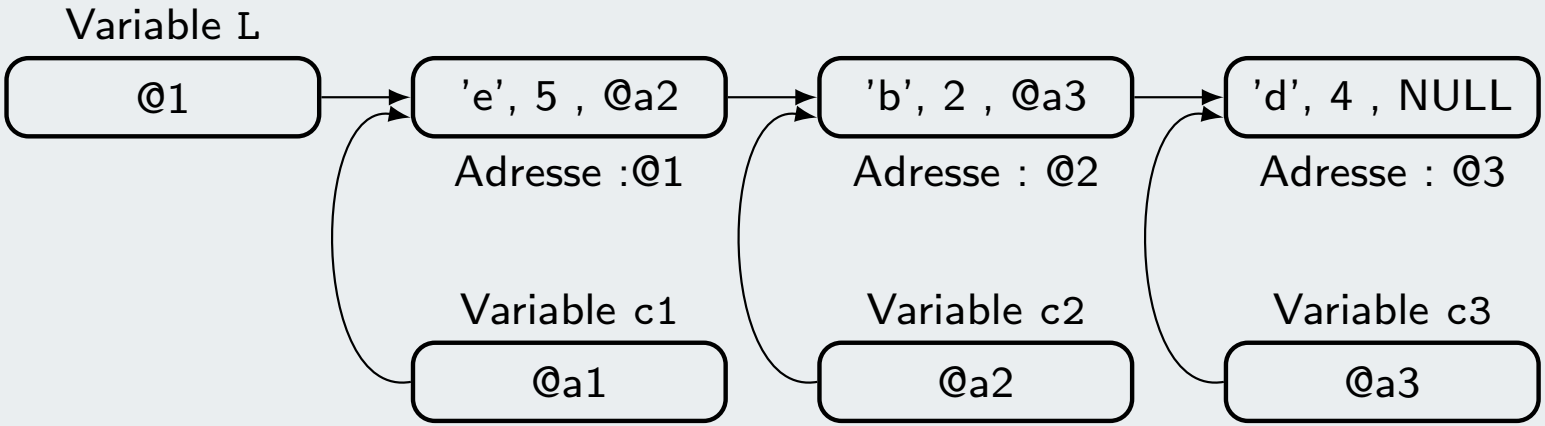
Pour une liste doublement chaînée, il faudrait ajouter un champ "prec".

Représentation avec une liste simplement chaînée

Exemple

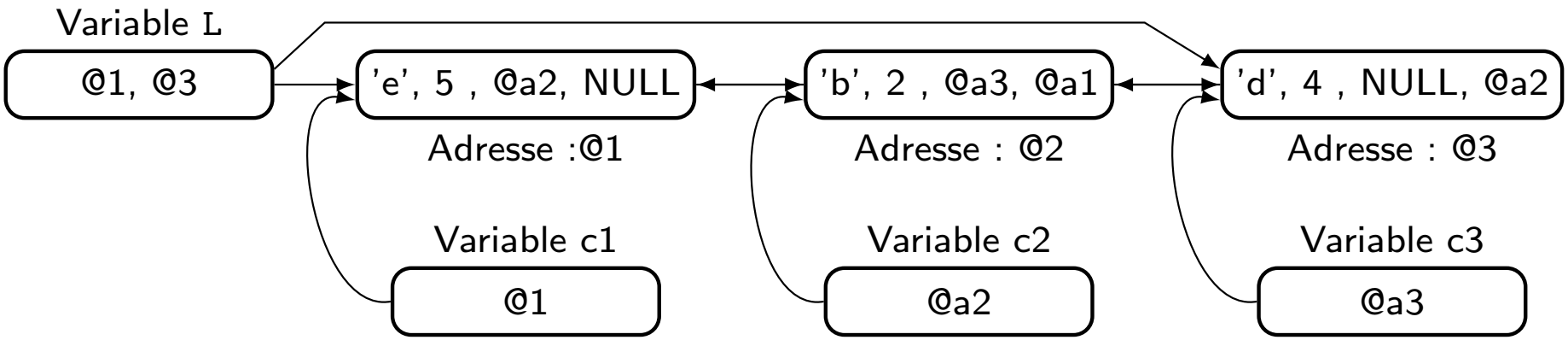
Construire une liste contenant ('e',5),('b',2), puis ('d',4).

```
1  Cellule* c1 = (Cellule*) malloc (sizeof(Cellule));
2  Cellule* c2 = (Cellule*) malloc (sizeof(Cellule));
3  Cellule* c3 = (Cellule*) malloc (sizeof(Cellule));
4  Cellule* L = c1;
5  c1->lettre='e';
6  c1->nombre=5;
7  c1->suiv=c2;
8  c2->lettre='b';
9  c2->nombre=2;
10 c2->suiv= c3;
11 c3->lettre='d';
12 c3->nombre=4;
13 c3->suiv=NULL;
```



Représentation avec une liste doublement chaînée

```
typedef struct tete { //On utilise une tete fictive
    Cellule* debut;
    Cellule* fin;
} TeteListe;
TeteListe* L = (TeteListe*) malloc (sizeof(TeteListe));
Cellule* c1 = (Cellule*) malloc (sizeof(Cellule));
Cellule* c2 = (Cellule*) malloc (sizeof(Cellule));
Cellule* c3 = (Cellule*) malloc (sizeof(Cellule));
L->debut=c1;
L->fin=c3;
c1->lettre='e';
c1->nombre=5;
c1->suiv=c2;
c1->prec=NULL;
c2->lettre='b';
c2->nombre=2;
c2->suiv= c3;
c2->prec=c1; //etc...
```



Quelques opérations sur les listes simplement chaînées

```c
/* Affiche le premier element de la liste L */
void affiche_debut(Cellule *L){
    if (L==NULL){
        printf("Liste vide\n");
    } else {
        printf("(%c,%d)\n",L->lettre,L->nombre);
    }
}
```

```c
/* Affiche le dernier element de la liste L */
void affiche_fin(Cellule *L){
    Cellule *cour;
    if (L==NULL){
        printf("Liste vide\n");
    } else {
        cour=L;
        while (cour->suiv!=NULL){
                cour=cour->suiv;
        }
        printf("(%c,%d)\n",cour->lettre,cour->nombre);
    }
}
```

Quelques opérations sur les listes simplement chaînées

```c
void affichage_des_elements(Cellule *L){
    Cellule* cour=L;
    while (cour!=NULL){
        printf("(%c,%d) ",cour->lettre, cour->nombre);
        cour=cour->suiv;
    }
    printf("\n");
}
```

```c
int possede_element(Cellule *L, char c, int i){
    Cellule* cour=L;
    while ((cour!=NULL)&&
                ((cour->lettre!=c)||(cour->nombre!=i))){
      cour=cour->suiv;
   }
   return (cour!=NULL);
}
```

```c
void suppression_au_debut(Cellule **L){
    Cellule *temp;
    if (*L!=NULL){
        temp=(*L)->suiv;
        free(*L);  /* Desallocation de la case supprimee */
        *L=temp;
    }
}
```

Quelques opérations sur les listes simplement chaînées

```
1  /* Suprime l'element (c,i) s'il existe */
2  void suppression_element(Cellule **L, char c, int i){
3      Cellule *cour;
4      Cellule *prec;
5      if (*L!=NULL){
6          cour=*L;
7          if ((cour->lettre==c) && (cour->nombre==i)){
8              suppression_au_debut(L);
9          }else{
10             prec=*L;
11             cour=cour->suiv;
12             while ((cour!=NULL) &&
13                       ((cour->lettre!=c)||(cour->nombre!=i))){
14                 prec=cour;
15                 cour=cour->suiv;
16             }
17             if (cour!=NULL){
18                 prec->suiv=cour->suiv;
19                 free(cour);
20             }
21         }
22     }
23 }
```

Quelques opérations sur les listes simplement chaînées

```c
void insertion_au_debut(Cellule **L, char c, int i){
    Cellule * nouv;
    nouv=(Cellule *) malloc(sizeof(Cellule));
    nouv->lettre=c;
    nouv->nombre=i;
    nouv->suiv=*L;
    *L=nouv; /* mise a jour de la case de debut */
}
```

```c
int main(){
    Cellule *L=NULL; /* Initialise la liste*/
    affichage_debut(L); // Affiche ``Liste vide''
    insertion_au_debut(&L,'A',3);
    insertion_au_debut(&L,'B',56);
    insertion_au_debut(&L,'C',43);
    affichage_debut(L); //Affiche (C,43)
    suppression_element(&L, 'B', 56);
    affichage_des_elements(L); //Affiche (C,43) (A,3)
    if (!possede_element(L,'D',10){
        insertion_au_debut(&L,'D',10);
    }
    return 0;
}
```

Complexité des opérations de la liste simplement chaînée

Pour une liste chaînée contenant n éléments.

Recherche/Accès

La recherche d'un élément est en $O(n)$ (dans le pire cas, on parcourt toute la liste). L'accès à un élément quelconque est en $O(n)$. L'accès à l'élément `début` est en $O(1)$, tandis que l'accès à l'élément `fin` est en $O(n)$.

Suppression d'un élément

Une fois l'élément à supprimer trouvée, l'opération se fait en $O(1)$. La suppression d'un élément quelconque est donc en $O(n)$, tandis que la suppression de l'élément début est en $O(1)$.

Insertion

L'insertion au début se fait en $O(1)$, tandis que l'insertion en fin se fait en $O(n)$.

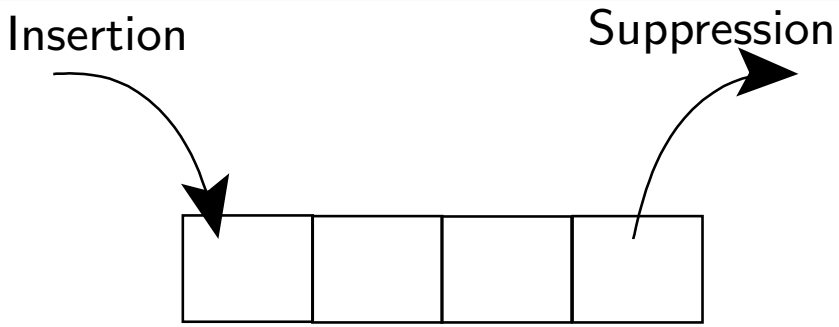
Remarque 1 : Le défaut de la liste simplement chaînée est son coût de l'accès à l'élément fin. Si cette opération est importante, il faut opter pour une liste doublement chaînée (c'est l'implémentation choisie par C++ et par java).

Remarque 2 : Pour garder un pointeur sur le dernier élément visité, on peut utiliser une liste circulaire, mais cela n'améliore pas les performances pire-cas.

File

Type de données : les files

Une file est un type de données représentant un ensemble de données homogènes, qui est fondée sur le principe du "premier entré, premier sorti" (aussi appelé FIFO en anglais pour "First In, First Out"). Dans une file, les premiers éléments ajoutés à la file seront les premiers à être retirés de la file. Ce type de données permet l'insertion aisée de nouvelles données, et l'extraction aisée de la plus ancienne donnée insérée.



Implémentation

En langage C, on peut implémenter une file comme on implémente une liste (doublement) chaînée.

Nombreuses applications : mémoires tampons, "pipe" sous linux, files d'attente, compilation, algorithmes divers (exemple : parcours en largeur).

Complexité des opérations sur les files

Opérations usuelles

Les opérations usuelles sur les files sont les suivantes :

- `creer_file` : crée une liste vide.
- `est_vide` : teste si une file est vide.
- `enfiler` : insérer un élément au début de file.
- `defiler` : supprimer l'élément à la fin de la file.

Complexité

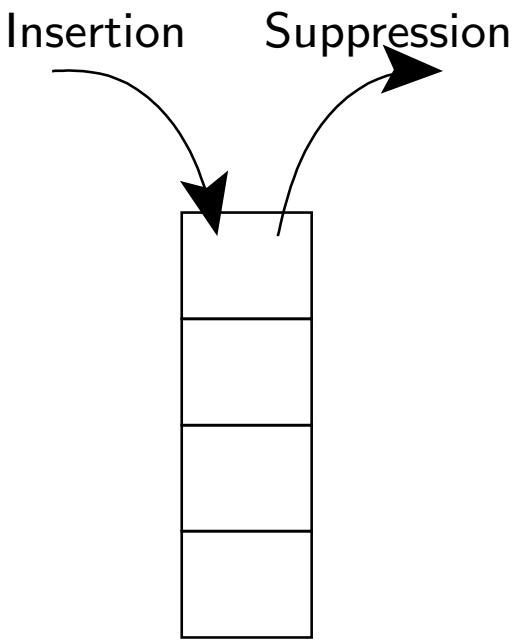
Avec une implémentation par liste doublement chaînée, toutes les opérations sont en $O(1)$. En effet, `enfiler` et `défiler` sont tous les deux en $O(1)$ car :

- on a un accès au premier et au dernier élément en $O(1)$.
- les fonctions *successeur* et *prédécesseur* permettent de "regrouper" deux morceaux en $O(1)$.

Pile

Type de données : les piles

Une pile est un type de données représentant un ensemble de données homogènes, qui est fondée sur le principe du "dernier entré, premier sorti" (aussi appelé LIFO en anglais pour "Last In, First Out"). Dans une pile, les derniers éléments ajoutés à la pile seront les premiers à être retirés. Ce type de données permet l'insertion aisée de nouvelles données, et l'extraction aisée des dernières données insérées.



Implémentation

En langage C, on peut implémenter une file avec un tableau C à une dimension, ou comme on implémente une liste (simplement) chaînée.

Nombreuses applications : Pile d'exécution, "undo/redo" dans un logiciel, algorithmes divers (exemple : parcours en profondeur).

Complexité des opérations sur les piles

Opérations usuelles

Les opérations usuelles sur les piles sont les suivantes :

- `creer_pile` : crée une pile vide.
- `est_vide` : teste si une file est vide.
- `empiler` : insérer un élément au début de la file.
- `depiler` : supprimer l'élément au début de la file.

Complexité

Avec une implémentation par liste chaînée ou avec un tableau C à une dimension, toutes les opérations sont en $O(1)$. En effet, avec une liste chaînée, on peut insérer un élément en tête en $O(1)$, puis récupérer le dernier élément ajouté en accédant au début de la liste, ce qui se fait aussi en $O(1)$. Avec un tableau C à une dimension, il suffit d'insérer les nouveaux éléments en fin de tableau, puis récupérer le dernier élément ajouté en accédant directement à la dernière case du tableau. La différence entre les deux implémentations est uniquement sur la réservation mémoire (contiguë ou non contiguë).

Retour sur la pile d'exécution

Appel de fonction et zone mémoire

Lors d'un appel à une fonction, une zone mémoire est réservée pour effectuer cet appel. Cette zone contient les variables déclarées dans la fonction, ainsi que les variables passées en paramètres. La zone mémoire n'est libérée que lorsque l'accolade fermante ou la commande `return` sont atteintes. Si une fonction `f` appelle une fonction g, alors la zone mémoire de `f` reste en place pendant l'exécution de g, et l'exécution de `f` reprend une fois g terminée.

Pile d'exécution

Le stockage et l'ordre des exécutions des fonctions sont gérées par une pile que l'on appelle la pile d'exécution. Les appels s'empilent dans l'ordre de leur appel et se dépilent dans l'ordre de leur terminaison. Entre temps, la mémoire n'est donc pas libérée ! Cela peut poser problème s'il y a beaucoup d'appels imbriqués, comme c'est le cas pour des appels récursifs.

Récursivité

Récursivité et zone mémoire

On appelle récursivité le fait qu'une fonction s'appelle elle-même. La récursivité utilise de la mémoire en grande quantité car les appels récursifs conservent une place en mémoire jusqu'à la fin de leur exécution. Par exemple, avec la fonction suivante, les appels ne seront terminés que lorsque le cas de base est atteint :

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

⇒ Quand cela est possible, on préfère utiliser une récursivité terminale.

Récursivité terminale

Une fonction est récursive terminale si l'appel récursif est directement retourné par la fonction appelante. Le retour de la fonction ne doit donc pas être le résultat d'une opération utilisant le retour de l'appel récursif. La récursivité terminale permet de limiter les appels récursifs qui sont stockés dans la pile d'exécution du programme.

Retour sur la fonction somme

Dans la première version, toutes les opérations “n+” sont stockées dans la pile. On dépile ces opérations une fois le cas de base atteint.

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

Dans la deuxième version, la fonction est récursive terminale, ce qui permet de ne pas saturer la pile d'exécution du programme avec toutes ces informations.

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

En plus de cette écriture terminale, il faut compiler avec l'option -02 de gcc pour que le compilateur optimise les récursivités.