

LE2 – Semestre 3

Structure de Données Dynamiques

support pour les travaux pratiques

Ce document est un **fil conducteur** pour les cinq premières séances.

Il permettra à chacun de progresser à son rythme.

Des polys de cours vous sont accessibles. Vous trouverez les liens dans la rubrique bibliographie.

Rappel 1 : Les pointeurs	page 2
Rappel 2 : Les structures	page 4
Rappel 3 : La récursivité	page 5
Rappel 4 : l'allocation dynamique de mémoire	page 8
Rappel 5 : argc et argv	page 9
Nouveauté 1 : les structures auto-référentes	page 10
Nouveauté 2 : allocation dynamique, les fonctions realloc et calloc	page 13
Nouveauté 3 : appels systèmes	page 14
TP1 La pile dynamique (<u>et contigue</u>)	page 15
TP2 La liste chaînée	page 30
bibliographie	page

Rappel 1 : les pointeurs

Lorsqu'on déclare une variable de type entier avec **int a** ; la machine réserve 4 octets en mémoire et cet emplacement mémoire est appelé **a**. Cet emplacement mémoire se situe à une certaine adresse de la mémoire. Cette adresse est notée **&a**. Cette adresse peut changer d'une exécution à l'autre.

Afficher **a** à l'écran, c'est afficher la valeur entière correspondant aux quatre octets évoqués.

Ces quatre octets sont remplis de 0 et de 1 désignant une valeur entière aléatoire.

En l'état, afficher **a** à l'écran, c'est afficher cette valeur entière aléatoire.

Il convient donc plutôt de déclarer et d'initialiser simultanément : **int a=2;** (ou toute autre valeur)

Lorsqu'on déclare un pointeur d'entier avec **int *pa** ; la machine réserve un nombre d'octets N pour y ranger une adresse. N dépend du processeur du pc utilisé. Comme **pa** n'est pas initialisé, cette boîte mémoire contient en l'état une valeur aléatoire désignant une adresse aléatoire de la mémoire. Il est donc vivement conseillé de déclarer et d'initialiser avec **int *pa = NULL;** ou **int *pa = &a;**

Afficher **pa** à l'écran, c'est afficher l'adresse contenue par cette variable : à savoir 0 dans le premier cas et l'adresse où se situe la variable **a** en mémoire dans le second cas.

Afficher ***pa** à l'écran, c'est afficher le contenu situé à l'adresse contenue par **pa**. Vous afficherez la valeur entière située à l'adresse zéro de la mémoire pour le premier cas, vous afficherez la valeur 2 stockée dans **a** dans le second cas.

Cas des fonctions et des paramètres déclarés en pointeurs :

```
void incremeter(int *pa)
{
    (*pa)++;
}

appel : int a=2; incremeter(&a);
```

L'adresse de la variable **a** est envoyée vers la fonction et la variable **pa** la reçoit. On dit que **pa** pointe sur **a**. Avec cette variable **pa**, vous devez écrire ***pa** pour accéder à la valeur contenue (2) à cette adresse. Cette valeur 2 est alors incrémentée et la variable **a** contient désormais la valeur 3.

Si vous tentez de modifier **pa** au sein de cette fonction (par **pa=0** ou **pa++** par exemple), le pointeur ne pointe plus sur **a**. Cela n'affecte en rien l'adresse de la variable **a**.

Autre cas :

```
void swap1(int *pa,int *pb)
```

```

{
    int x=*pa; *pa=*pb; *pb=x;
}

appel : int a=2; int b=3; swap1(&a,&b);

```

Cette fonction permute les valeurs de a et b. a vaudra 3 et b vaudra 2.

```

void swap2(int **ppa,int **ppb)
{
    int *x=*ppa; *ppa=*ppb; *ppb=x;
}

appel : int a=2; int b=3; int *pa=&a; int *pb=&b; swap2(&pa,&pb);

```

Cette fonction ne permute pas les valeurs de a et b mais permute les adresses contenues dans pa et pb. pa pointera vers la variable b et pb pointera vers la variable a.

Rappel 2 : les structures

Nous avons déjà utilisé les structures l'an dernier. Voici quelques rappels, rappelez-vous la structure T_Livre :

```
typedef char T_titre[60];  
typedef char T_auteur[30];  
typedef struct {  
    T_titre leTitre;  
    T_auteur lAuteur;  
}T_Livre;
```

Cette structure possède 2 champs et peut être utilisée de deux manières :

1. En déclarant **T_Livre monLivre;** on pourra accéder aux champs par la notation **pointée** :
monLivre.leTitre ou **monLivre.lAuteur**
2. En déclarant un pointeur **T_Livre *pMonLivre;** et en l'initialisant avec
pMonLivre=& monLivre on pourra accéder aux champs par la notation **fléchée** :
pMonLivre->leTitre ou **pMonLivre->lAuteur**

Rappel 3 : la récursivité

Il y a récursivité lorsque qu'une fonction fait appel à elle-même.

Décortiquons ensemble l'exemple classique du calcul d'un terme de rang **n** de la suite de Fibonacci dont voici la fonction :

```
int fibo(int n)
{
    if (n==0) return 0;
    if (n==1) return 1;
    return fibo(n-1)+fibo(n-2);
}
```

Cette fonction est appelée par : **int valeur = fibo(4);**

Suivons le déroulement des appels :

- La variable n va recevoir cette valeur 4 : Renommons la variable n en n1. Comme n1 est différent de 0 et de 1, la ligne suivante retournera fibo(3)+fibo(2). Il faut d'abord évaluer fibo(3) avant d'évaluer fibo(2). L'exécution de la fonction fibo(4) n'est donc pas achevée totalement qu'un nouvel appel à cette fonction est demandé par fibo(3). Lorsque cette valeur sera connue, il faudra enchaîner avec fibo(2) pour enfi pouvoir renvoyer fibo(3)+fibo(2) et ainsi terminer cette exécution n°1.
- appel n°2 : fibo(3). Renommons la variable n en n2 et elle reçoit 3. Comme n2 est différent de 0 et de 1, la ligne suivante retournera fibo(2)+fibo(1). Il faut d'abord évaluer fibo(2) avant d'évaluer fibo(1). L'exécution de la fonction fibo(3) n'est donc pas achevée totalement qu'un nouvel appel à cette fonction est demandé par fibo(2). Lorsque cette valeur sera connue, il faudra enchaîner avec fibo(1) pour enfin pouvoir renvoyer fibo(2)+fibo(1) et ainsi terminer cette exécution n°2.
- appel n°3 : fibo(2). Renommons la variable n en n3 et elle reçoit 2. Comme n3 est différent de 0 et de 1, la ligne suivante retournera fibo(1)+fibo(0). Il faut d'abord évaluer fibo(1) avant d'évaluer fibo(0). L'exécution de la fonction fibo(2) n'est donc pas achevée totalement qu'un nouvel appel à cette fonction est demandé par fibo(1). Lorsque cette valeur sera connue, il faudra enchaîner avec fibo(0) pour enfin pouvoir renvoyer fibo(1)+fibo(0) et ainsi terminer cette exécution n°3.
- appel n°4 : fibo(1). Renommons la variable n en n4 et elle reçoit 1. Comme n4 vaut 1, cette appel n°4 va se terminer proprement par un return 1. C'est donc la première exécution qui se termine totalement. Cette valeur 1 est renvoyée au niveau de l'exécution n°3 que nous avons quittée ci-dessus. (exécution n°4 TERMINEE)

- Retour à l'appel n°3 : fibo(1) reçoit donc la valeur 1 et dans cette exécution n°3, il faut encore évaluer la valeur de fibo(0).
 - appel n°5 : fibo(0). Renommons la variable n en n5 et elle reçoit 0. Comme n5 vaut 0, cette appel n°5 va se terminer proprement par un return 0. C'est donc la deuxième exécution qui se termine totalement. Cette valeur 0 est renvoyée au niveau de l'exécution n°3 que nous avons quittée ci-dessus. (exécution n°5 TERMINEE)
 - Retour à l'appel n°3 : fibo(0) reçoit donc la valeur 0 et dans cette exécution n°3, le return final peut avoir lieu car $\text{fibo}(1) + \text{fibo}(0)$ est désormais connu et vaut $1 + 0 = 1$. (exécution n°3 TERMINEE). Cette valeur va être renvoyée à l'appel intervenu dans l'exécution n°2.
- Retour à l'appel n°2 : fibo(2) reçoit donc la valeur 1 et dans cette exécution n°2, le return final ne peut pas avoir lieu car il faut encore évaluer fibo(1).
 - appel n°6 : fibo(1). Renommons la variable n en n6 et elle reçoit 1. Comme n6 vaut 1, cette appel n°6 va se terminer proprement par un return 1. Cette exécution se termine totalement. Cette valeur 1 est renvoyée au niveau de l'exécution n°2 que nous avons quittée ci-dessus. (exécution n°6 TERMINEE)
- Retour à l'appel n°2 : fibo(1) reçoit donc la valeur 1 et dans cette exécution n°2, le return final peut maintenant avoir lieu car $\text{fibo}(2) + \text{fibo}(1)$ vaut $1 + 1 = 2$. (exécution n°2 TERMINEE). Cette valeur va être renvoyée à l'appel intervenu dans l'exécution n°1.
- Retour à l'appel n°1 : fibo(3) reçoit donc la valeur 2 et dans cette exécution n°1, le return final ne peut avoir lieu car fibo(2) n'est pas connu dans ce contexte.
 - appel n°7 : fibo(2). Renommons la variable n en n7 et elle reçoit 2. Comme n7 est différent de 0 et de 1, la ligne suivante retournera $\text{fibo}(1) + \text{fibo}(0)$. Il faut d'abord évaluer fibo(1) avant d'évaluer fibo(0). L'exécution n°7 n'est donc pas achevée totalement qu'un nouvel appel à cette fonction est demandé par fibo(1). Lorsque cette valeur sera connue, il faudra enchaîner avec fibo(0) pour enfin pouvoir renvoyer $\text{fibo}(1) + \text{fibo}(0)$ et ainsi terminer cette exécution n°7.)
 - appel n°8 : fibo(1). Renommons la variable n en n8 et elle reçoit 1. Comme n8 vaut 1, cette appel n°8 va se terminer proprement par un return 1. Cette exécution se termine totalement. Cette valeur 1 est renvoyée au niveau de l'exécution n°7 que nous avons quittée ci-dessus. (exécution n°8 TERMINEE)
 - Retour à l'appel n°7 : fibo(1) reçoit donc la valeur 1 et dans cette exécution n°7, le return final ne peut pas avoir lieu car il faut encore évaluer fibo(0).
 - appel n°9 : fibo(0). Renommons la variable n en n9 et elle reçoit 0. Comme n9 vaut 0, cette appel n°9 va se terminer proprement par un return 0. Cette exécution se

termine totalement. Cette valeur 0 est renvoyée au niveau de l'exécution n°7 que nous avons quittée ci-dessus. (exécution n°9 TERMINEE)

- Retour à l'appel n°7 : $\text{fib}(1) + \text{fib}(0)$ est maintenant connu et vaut $1 + 0 = 1$. Le return final peut avoir lieu. La valeur 1 est renvoyée à l'appel 1. (exécution n°7 TERMINEE)
- Retour à l'appel n°1 : $\text{fib}(2)$ reçoit donc la valeur 1 et dans cette exécution n°1, le return final peut ENFIN. L'exécution n°1 s'achève donc et le résultat renvoyé est $2 + 1 = 3$. $\text{fib}(4)$ vaut donc 3 et la variable valeur reçoit alors 3. (exécution n°1 TERMINEE)

Conclusion : La fonction a été appelée 9 fois. Nous pouvons apprécier l'importance des 2 conditions d'arrêt (if ($n == 0$) et if ($n == 1$)).

Rappel 4 : l'allocation dynamique de mémoire

La fonction **malloc** a été utilisée l'an dernier. En voici le prototype :

void * malloc(size_t memorySize);

voir <https://koor.fr/C/cstdlib/malloc.wp>

Pourquoi void * ?

Tout simplement car lors de son utilisation, la fonction malloc ne sait pas de quoi elle renverra l'adresse. C'est au développeur de le spécifier.

Ainsi, l'instruction suivante permet d'allouer 10 entiers de 4 octets et de stocker l'adresse du premier d'entre eux dans le pointeur déclaré pour cela :

int *pi=NULL; pi = (int *) malloc(10 * sizeof(int));

On dit alors que pi pointe sur le premier octet des quarante octets réservés.

Attention , écrire l'instruction pi++; après ce malloc, déplace pi sur l'entier suivant. L'adresse n'a pas été incrémentée , elle a été augmentée de 4 octets (car int ici). Elle le serait de 8 dans le cas d'un pointeur de double.

On peut considérer que l'espace mémoire de 40 octets constitue en quelque sorte, un tableau de 10 entiers. Ainsi, les instructions suivantes sont équivalentes :

***pi=99; ou pi[0]=99;**

***(pi+1)=100; pi[1] =100;**

Rendre à la machine l'espace mémoire alloué se fait grâce à l'instruction **free**. En voici le prototype :

void free(void * pointer);

voir <https://koor.fr/C/cstdlib/malloc.wp>

Il faut donc écrire l'instruction suivante pour libérer les 40 octets et les rendre à nouveau disponibles pour une autre utilisation :

free(pi);

ATTENTION : Si vous deviez perdre de façon malencontreuse, l'adresse contenue par pi (exemple: avec une simple affectation malheureuse telle que **pi=NULL;**), votre programme monopolisera l'espace de 40 octets jusqu'à la fin de son exécution (et sans pouvoir l'utiliser durant ce temps).

Vous trouverez plus loin dans ce document, une présentation de deux autres fonctions (realloc et calloc).

Rappel 5 : argc et argv

Nous avons déjà rencontré par le passé, cette déclaration de la fonction main :

```
int main(int argc , char *argv[] )
```

Parfois, il est possible de trouver une déclaration équivalente qui est :

```
int main(int argc , char **argv )
```

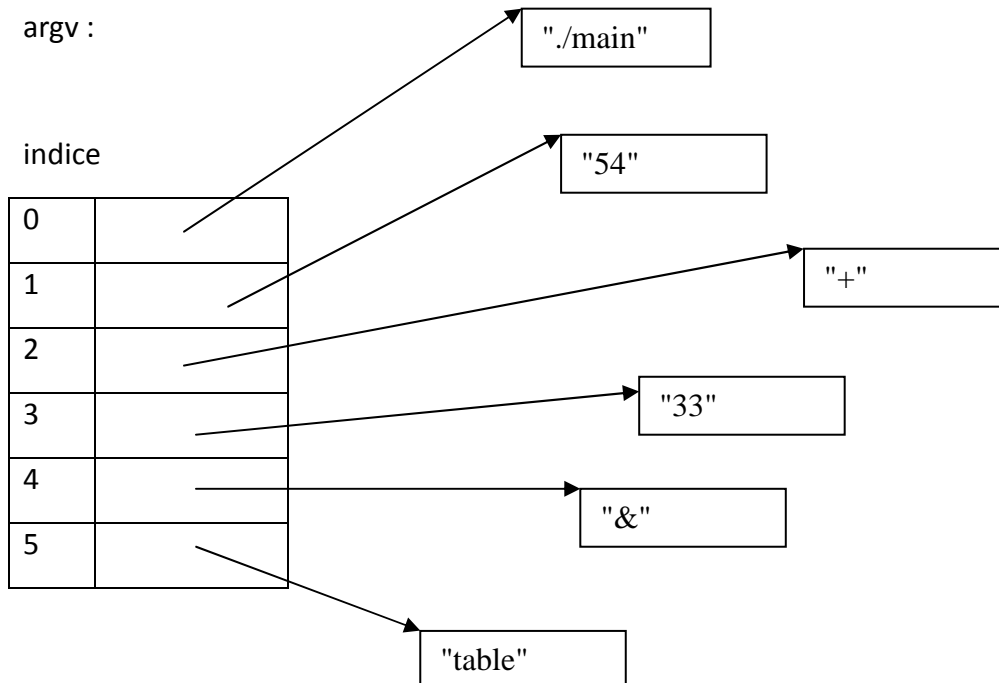
Les paramètres argc et argv servent à transmettre des données lors du lancement du programme.

Quel sera le contenu de argc et argv si la ligne de commande suivante est demandée :

```
./main 54 + 33 & table
```

Dans ce cas, **argc** vaut **6** qui représente le nombre de morceaux constituant cette commande.

Concernant argv qui est un tableau d'adresse de chaînes de caractères, il sera dimensionné avec cette valeur de argc et contiendra dans chacune de ses cases, l'adresse mémoire de chacune des sous chaînes constituant la commande. Le dessin ci dessous illustre la situation :



Nouveauté 1 : les structures auto-référentes ou structures récursives

Il s'agit d'une structure dont l'un ou plusieurs de ses champs est/sont déclaré(s) du même type que la structure elle-même. Exemple :

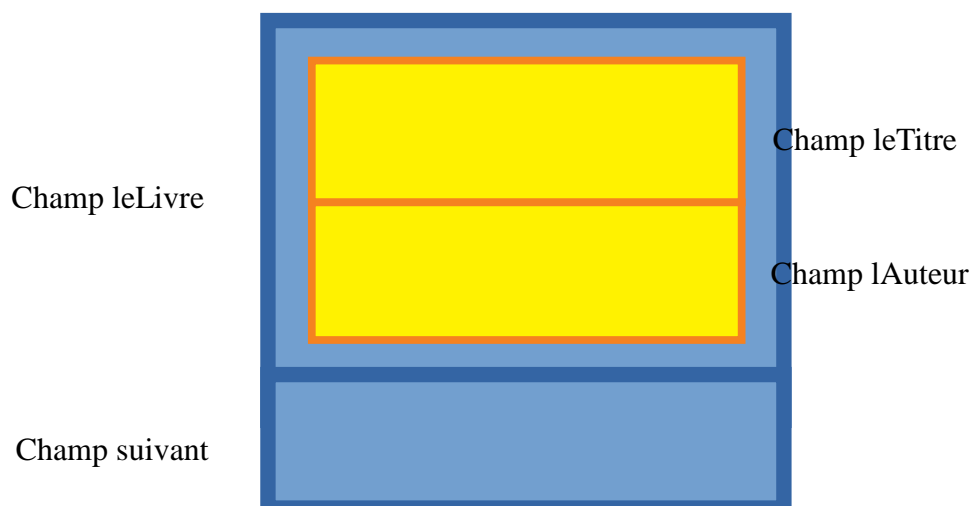
```
typedef char T_titre[60];  
typedef char T_auteur[30];  
typedef struct {  
    T_titre leTitre;  
    T_auteur lAuteur;  
    }T_Livre;
```

```
typedef struct maille {  
    T_Livre leLivre;  
    struct maille *suivant; //ici se trouve l'auto  
référence  
}T_Maille ;
```

A noter qu'après le mot clé **struct**, apparaît un nom intermédiaire (**maille**) permettant d'y faire référence dans l'un des champs alors que le type T_Maille n'est pas encore totalement déclaré.

Le champ nommé **suivant** permet alors de mémoriser l'adresse d'une autre maille.

Une variable déclarée de type T_Maille (T_Maille uneMaille;) peut alors être représentée par ce schéma :



Pour utiliser cette structure, il faudra écrire des choses telles que :

```
strcpy( uneMaille.leLivre.leTitre,"GERMINAL")
```

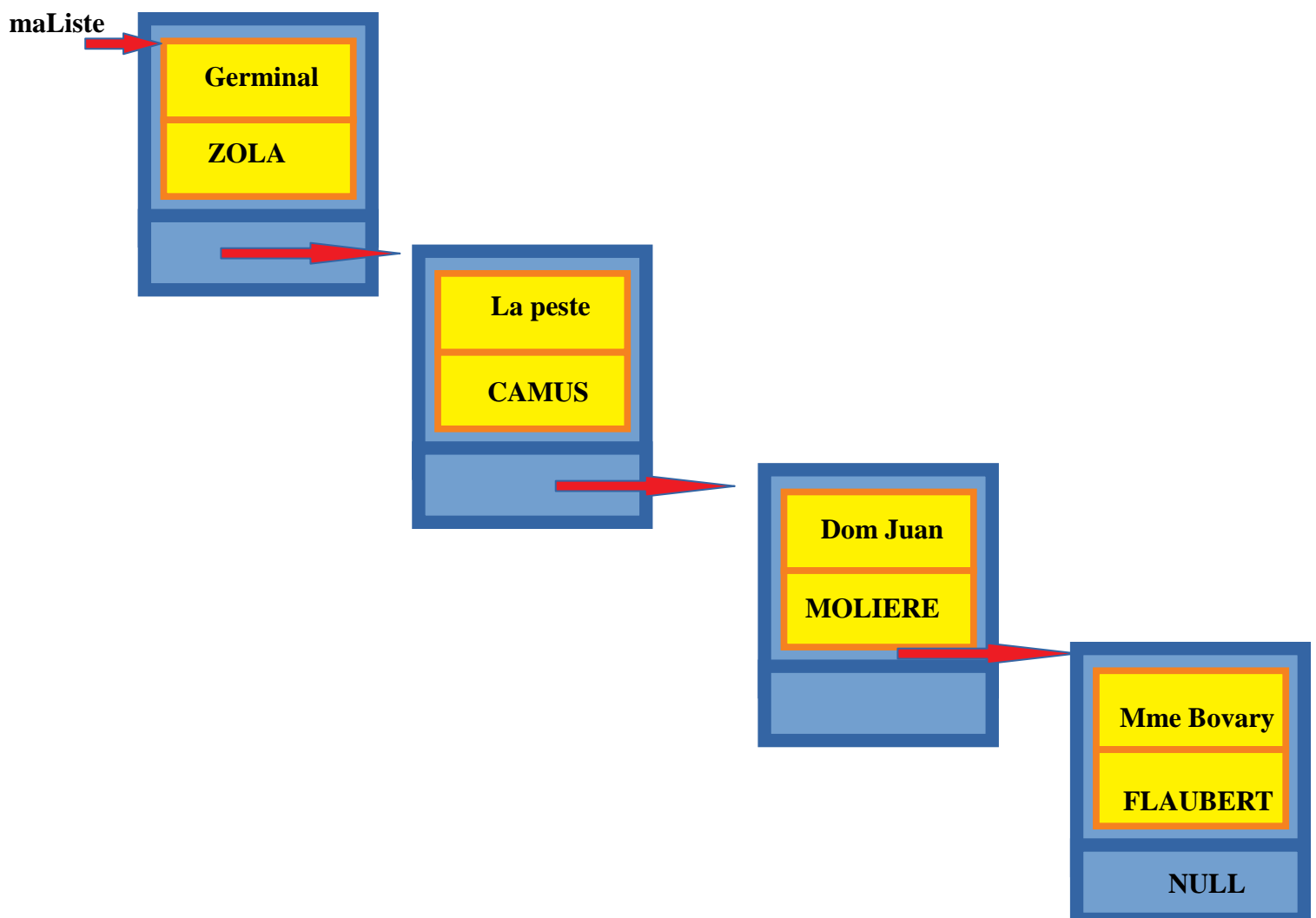
```
strcpy( uneMaille.leLivre.lAuteur,"ZOLA");;
```

et si on déclarait maintenant **T_Maille uneAutreMaille;** , on pourrait alors écrire : :

```
uneMaille.suivant = &uneAutreMaille;
```

Ainsi né le concept de liste chaînée (pensez à une chaîne de vélo) où chaque maille est reliée à une autre maille (sa suivante) et où la dernière maille contient l'adresse de la première (voir figure ci dessous).

Illustration d'une liste chaînée NON CIRCULAIRE de 4 mailles :



NULL symbolise la fin de la liste chaînée.

Ma variable nommée `maListe` est un pointeur de maille et contient l'adresse de la première maille de la liste chaînée.

Pour une liste circulaire, il suffit de remplacer NULL par l'adresse de la première maille.

Remarque : une liste chaînée ne connaît pas sa taille (son nombre de maille). Dans le cas d'une liste non circulaire, pour la parcourir complètement, il faut donc aller de maille en maille jusqu'à détection de NULL (marqueur de fin de liste).

Dans les deux cas précédent (liste simple ou circulaire), on parle de liste **UNILATERALE** (car un seul sens de parcours). En effet, la structure ne dispose que d'un seul champ nommé **suivant** qui mémorise l'adresse de la maille suivante.

En ajoutant dans la déclaration de la structure un champ supplémentaire déclaré et nommé par

```
typedef struct maille {  
    T_Livre leLivre;  
    struct maille *suivant; //ici se trouve l'auto référence  
    struct maille *precedent; //auto référence aussi  
    }T_Maille ;
```

nous pourrions alors parler de liste chaînée BILATERALE (avec deux sens de parcours de la liste).

L'avantage d'une telle structure de données est que contrairement à la pile pour laquelle les éléments sont tous stockés dans une zone de mémoire contigue, la liste chaînée abandonne cette contrainte qui peut s'avérer sévère lorsque la mémoire est fortement occupée. En d'autres mots, cela signifie que si la mémoire est fortement occupée, il se pourrait qu'une allocation dynamique d'un espace important devienne impossible.

Pour palier à ce défaut, une liste chaînée possède un avantage. En effet, elle n'est qu'un ensemble non contigu de mailles. Il suffit de mémoriser l'adresse de la première maille pour retrouver toutes les autres mailles. Les mailles peuvent alors être parsemées dans la mémoire, nous saurons les retrouver.

Nouveauté 2 : allocation dynamique, les fonctions realloc et calloc

Nous avons essentiellement utilisé la fonction **malloc** pour allouer de la mémoire.

Il existe deux autres fonctions pour atteindre cet objectif :

1. la fonction **calloc** dont le prototype est le suivant :

```
void * calloc( size_t elementCount, size_t elementSize );
```

Cette fonction alloue un bloc de `elementCount * elementSize` octets et les initialise à 0 (ce que ne fait pas le `malloc`). Elle renvoie l'adresse de ce bloc.

2. la fonction **realloc** dont le prototype est le suivant :

```
void * realloc( void * pointer, size_t memorySize );
```

Cette fonction permet de réallouer un bloc de mémoire (à la hausse ou à la baisse en taille) à l'adresse désignée par **pointer**.

Dans le cas d'un agrandissement de taille, cela veut dire que si l'espace mémoire libre qui suit le bloc initial à réallouer est suffisamment grand, le bloc de mémoire est simplement agrandi et **pointer** ne change pas. Cette adresse **pointer** est tout de même renvoyée (même si dans ce cas, elle ne change pas).

Par contre, si cet espace libre n'est pas suffisant, un nouveau bloc de mémoire sera alloué ailleurs en mémoire, le contenu de la zone d'origine recopié dans la nouvelle zone et le bloc mémoire d'origine sera libéré automatiquement. Dans ce cas, l'adresse renvoyée par `realloc` est cette nouvelle adresse du nouveau bloc.

Nouveauté 3 : si une erreur apparaît lors d'un appel système : **errno**, **strerror** et **perror**

Les appels systèmes positionnent en cas d'erreur un code d'erreur dans la variable `errno`. Vous lirez la documentation sur le fichier d'entête `errno.h` pour en savoir davantage.

Ce code est associé à un message qu'il est possible de connaître à l'aide de la fonction suivante:

`char * strerror(int code)`

La fonction `perror` affiche ce message associé à la valeur actuelle de `errno`. Ce message système sera précédé de la chaîne fournie en paramètre.

`void perror(const char *chaîne)`

Séance 1

TP1 La pile dynamique (contigue)

Introduction :

En partant des codes sources du TP9 SDS1 de l'an dernier, ce document vous guidera vers la mise en oeuvre d'une pile dynamique.

Les objectifs successifs que vous trouverez ci-dessous, vous feront franchir étape par étape les difficultés.

Ce document est un donc fil rouge à suivre.

Le dernier objectif est un exercice d'application utilisant la pile dynamique que vous aurez élaborée.

Objectif 1 : passer d'une pile statique contigue à une pile dynamique contigue : les déclarations

Rappel : le terme contigue signifie que l'espace mémoire réservé à nos piles est un espace continu d'octets. Ce bloc mémoire n'est en aucun cas morcelé. Pour des piles de très grandes taille, l'allocation mémoire pourrait devenir problématique. C'est pour cela que nous nous tournerons vers la notion de liste chaînée au TP2.

1. Créez un nouveau répertoire nommé **tp_piledyn** et placez vous dans celui-ci.
2. Commencez par cloner les codes sources de l'an dernier :
git clone <https://github.com/kub62/tp9>
3. Placez vous dans le répertoire **tp_piledyn** créé.
4. Dans le fichier pile.h, vous trouvez cette ancienne déclaration d'une pile statique:

```
typedef struct
{
    T_Elt Elts[MAX]; // stockage d'éléments de la case 0 à la case MAX-1
    int nbElts;
} T_Pile;
```

Ajoutez celle-ci :

```
typedef struct
{
    T_Elt *Elts; // pointeur sur un élément
    int nbElts;
} T_PileD; //type pile dynamique
```

5. Remplacez T_Pile par **T_PileD** partout dans pile.h et pile.c (ainsi que dans votre main)
6. Compilez avec **make**. Vous devez constater des warnings mais pas d'erreur
7. La macro constante MAX ne sert plus à rien : supprimez la.

Objectif 2 : passer d'une pile statique à une pile dynamique : initialisation de la pile dynamique

1. Dans pile.h, ajoutez :

```
#include <assert.h>

#include <stdlib.h>

#define SEUIL 5
```

2. Dans pile.c, modifier la fonction initPile :

```
void initPile( T_PileD * P)
{
    P->Elts=(T_Elt *)malloc(SEUIL*sizeof(T_Elt));

    P->nbElts=0; //important

    assert(P->Elts);

    printf("Initialisation reussie de la pile dynamique");

}
```

La fonction **assert** agit comme une levée d'exception en Java. Ici, si P→Elts vaut null après le malloc , la fonction initPile sera interrompue et le printf , pas exécuté.

Si le malloc a renvoyé une adresse non nulle à P→Elts , la fonction se terminera normalement.

Pour en savoir + : <https://koor.fr/C/cassert/assert.wp>

Pour en savoir + sur malloc : <https://koor.fr/C/cstdlib/malloc.wp>

3. Dans tp9.c, écrivez une fonction testPile qui fera appel à initPile.



```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp9
File Edit View Search Terminal Help
kub@kub-HP-ProBook-650-G2:~/SDA/tp9$ ./tp9

SDA1 TP9

1 : tester mon fichier file.c
2 : tester mon fichier pile.c
3 : afficher et compter les permutations d'une chaine
4 : afficher et compter les solutions pour un échiquier
0 : QUITTER
votre choix ? 2
Initialisation réussie de la pile dynamique
```

4. Compilez, testez. Vous devez constater la situation suivante :
Lors de l'initialisation de la pile, SEUIL emplacements consécutifs de mémoire sont alloués.
Chaque emplacement mesure sizeof(T_Elt) octets.
5. Profitons-en à ce stade pour renommer les choses :
 1. tp9.c devient tp1sdd2.c (modifiez aussi l'intérieur de ce fichier : le commentaire de la première ligne et le premier printf du menu)
 2. modifiez le makefile en conséquence (remplacez partout tp9 par tp1sdd2)
 3. le répertoire tp9 peut aussi être renommé
6. Compilez, testez. Vous devez constater la situation suivante :

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help
kub@kub-HP-ProBook-650-G2:~/SDA/tp1sdd2$ ./tp1sdd2

SDD2 TP1

1 : tester mon fichier file.c
2 : tester mon fichier pile.c
3 : afficher et compter les permutations d'une chaîne
4 : afficher et compter les solutions pour un échiquier
0 : QUITTER
votre choix ? 2
Initialisation réussie de la pile dynamique
```

Objectif 3 : passer d'une pile statique à une pile dynamique : pilePleine et empiler (push)

Bien qu'une pile dynamique ne soit jamais pleine conceptuellement, elle peut l'être en réalité après l'insertion de $n \cdot \text{SEUIL}$ éléments.

Quand $n=1$, l'insertion du cinquième (car pour nous $\text{SEUIL}=5$) élément rend la pile pleine. Il faut alors agrandir l'espace alloué précédemment grâce à la fonction **realloc**.

La fonction pilePleine ne renverra donc jamais 1.

1. Complétez la fonction pilePleine avec :

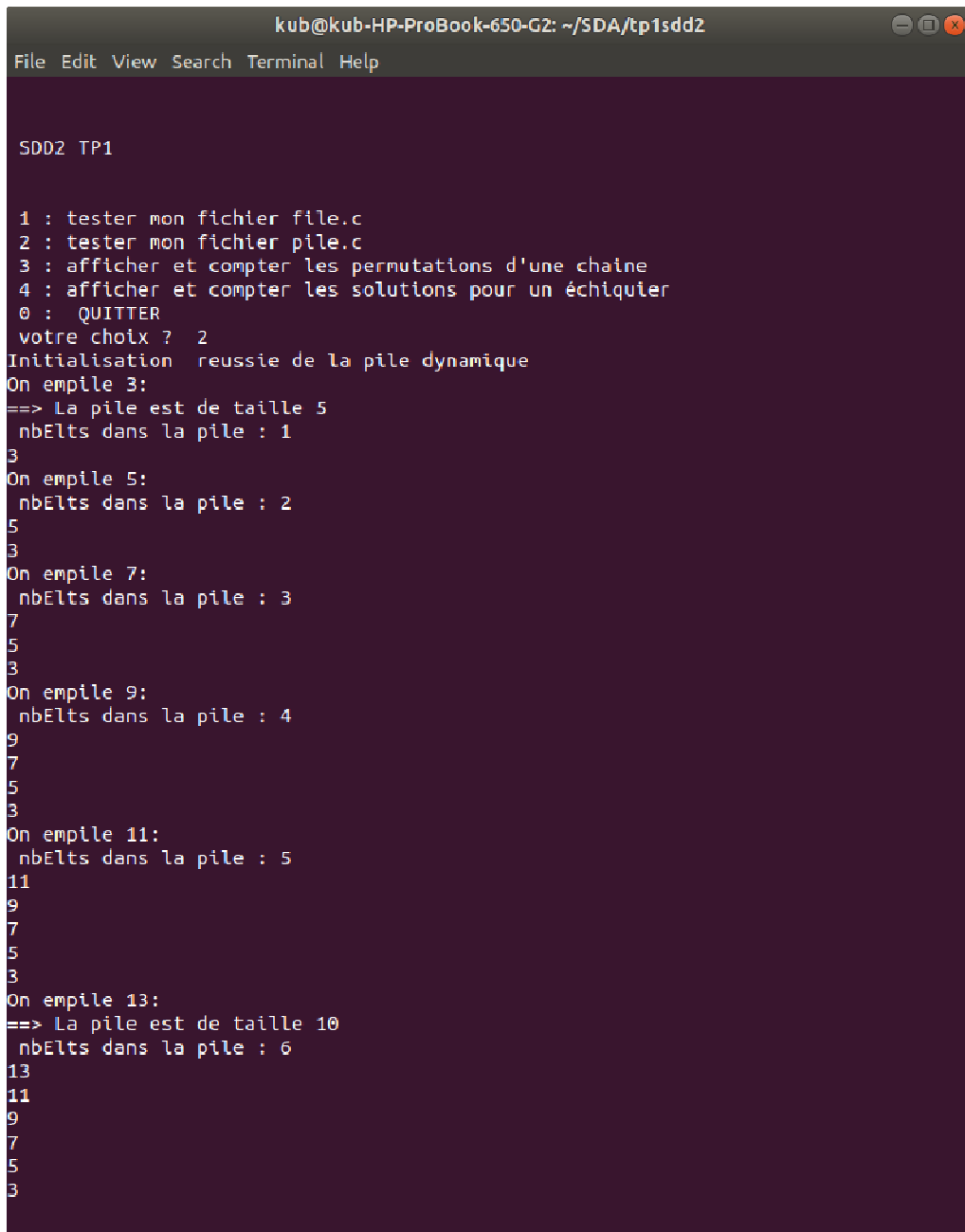
```
int pilepleine( T_PileD *P)
{
    int nouvelleTaille;

    if (P->nbElts%SEUIL==0) {
        nouvelleTaille=((P->nbElts/SEUIL)+1)*SEUIL;
        P->Elts=realloc(P->Elts,nouvelleTaille*sizeof(T_Elt));
        printf("\nn==> La pile est de taille %d ",nouvelleTaille);
    }
}
```

```
assert(P->Elt);
```

```
return 0; }
```

2. A l'aide des fonctions **testPile** et **empiler**, empilez jusqu'à six éléments en vérifiant par des affichages le dimensionnement de la pile :



```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help

SDD2 TP1

1 : tester mon fichier file.c
2 : tester mon fichier pile.c
3 : afficher et compter les permutations d'une chaîne
4 : afficher et compter les solutions pour un échiquier
0 : QUITTER
votre choix ? 2
Initialisation réussie de la pile dynamique
On empile 3:
==> la pile est de taille 5
nbElt dans la pile : 1
3
On empile 5:
nbElt dans la pile : 2
5
3
On empile 7:
nbElt dans la pile : 3
7
5
3
On empile 9:
nbElt dans la pile : 4
9
7
5
3
On empile 11:
nbElt dans la pile : 5
11
9
7
5
3
On empile 13:
==> la pile est de taille 10
nbElt dans la pile : 6
13
11
9
7
5
3
```

ici, on empile successivement 3 puis 5 puis 7 puis 9 puis 11 puis 13

On constate que la pile est agrandie au sixième élément

A ce stade, nous pouvons empiler des éléments sans limite (sauf limite mémoire du PC). Notre pile ne sera jamais pleine.

Objectif 4 : passer d'une pile statique à une pile dynamique : pileVide et dépiler(pop)

1. La notion de pile vide ne varie pas d'une pile statique vue l'an dernier. Il faut que le nombre d'élément soit égal à 0 et cette fonction renvoie dans ce cas 1.
2. Concernant la fonction dépiler, il faut penser à réduire la taille occupée en mémoire lorsque l'élément dépilé est le premier d'un SEUIL de mémoire. Plus clairement, notre SEUIL étant égal à 5, lorsque nous dépilons le 6ème ou le 11ème, etc... il faut libérer le SEUIL de mémoire inoccupé. Après avoir lu et compris la fonction ci-dessous, introduisez la dans votre code.

```
int depiler( T_PileD *P, T_Elt *pelt) //renvoie 0 si pile vide, sinon 1
{
    int nouvelleTaille;

    if (!pilevide(P))
    {
        *pelt=P->Elts[--P->nbElts];

        if (P->nbElts>0 && P->nbElts%SEUIL==0)
        {
            nouvelleTaille=(((P->nbElts)/SEUIL))*SEUIL;
            P->Elts=realloc(P->Elts,nouvelleTaille*sizeof(T_Elt));
            printf("\n==> La pile est de taille %d ",nouvelleTaille);
            assert(P->Elts);
        }

        return 1;
    }

    return 0;
}
```

3. A l'aide des fonctions **testPile** et **depiler**, empilez onze éléments puis dépilez en au moins six en vérifiant par des affichages le bon dimensionnement de la pile. Finissez par dépiler douze éléments pour détecter la pile vide.

A ce stade, nous pouvons empiler et dépiler des éléments (int) dans une pile dynamique.

Objectif 5 : passer d'une pile statique à une pile dynamique : avec une fonction d'affichage respectueuse du fonctionnement de la pile

Nous nous forçons à respecter le principe fondamental LIFO de la pile. Pour afficher totalement la pile, il nous faut la dépiler totalement, stocker ses éléments dans une pile temporaire puis reconstituer la pile dans son état initial.

Objectif 6 : passer d'une pile statique à une pile dynamique : changer la nature du type T_Elt

Si votre librairie de gestion de pile dynamique est bien écrite, vous n'aurez pas à retoucher le code de `pile.c/pile.h`.

1. Dans `es.h`, modifier le type `T_Elt` et remplacez `int` par `float`.
2. Dans `es.c`, remplacez les `%d` par `%f`
3. Dans la fonction `testPile`, remplacez aussi les `%d` par des `%f`, remplacez les valeurs entières empilées jusqu'ici par des réels.
4. Compilez sans le `makefile`

Objectif 7 : mise en oeuvre de la pile dynamique : exercice d'application n°1

La **notation polonaise inverse** (Reverse Polish Notation **RPN**) est une forme différente d'écrire les expressions mathématiques. Elle présente des avantages que vous pourrez retrouver ici :

https://fr.wikipedia.org/wiki/Notation_polonaise_inverse

Une calculatrice RPN est disponible ici :

http://www.ac-grenoble.fr/ugine/m/docs/calculatrice_rpn.html

Cet exercice d'application consiste à utiliser notre pile dynamique pour transformer une expression mathématique classique en son équivalent RPN.

Vous empilerez donc des char (modifiez de suite le type T_Elt).

Exemple 1: $(3+4)*(5-2)$ doit devenir **34+52-***

Exemple 2: $(a+b*c)-(a*d)$ doit devenir **abc*+ad*-**

Un opérande est un chiffre ou une lettre

Un opérateur est un signe par + - * /

Voici l'algorithme général en pseudo code (on suppose que l'expression à traduire ne comporte que des paires de parenthèse () , des opérandes et des opérateurs) :

Tant qu'il y a des caractères à lire sur l'expression à traduire {

1. Prendre le caractère suivant de l'expression

2. Si c'est un opérande alors

l'afficher à l'écran

sinon Si c'est une parenthèse ouvrante alors

empiler cette parenthèse

sinon Si c'est un opérateur, alors

1. Si la pile est vide alors empiler l'opérateur

2. Si le sommet de la pile est une parenthèse

ouvrante alors empiler l'opérateur

3. Si l'opérateur est prioritaire sur celui au sommet de la pile alors empiler l'opérateur

sinon

dépiler l'opérateur et l'afficher à l'écran

empilez l'opérateur courant

5. Si c'est une parenthèse fermante, dépiler et afficher les opérateurs jusqu'à ce que l'on rencontre la parenthèse ouvrante qui sera dépilée aussi.

}

Enlever tous les opérateurs restants et les placer sur le fichier de sortie.

Comment y parvenir ?

Voici la manière de procéder en lisant l'expression de l'exemple 1 considérée sous forme de chaîne de caractères sans espace "(3+4)*(5-2)" :

1. prendre un pile vide puis démarrer la lecture de l'expression à traduire
2. lecture de (, c'est une parenthèse ouvrante, on l'empile
3. lecture de 3 , c'est un opérande, on l'affiche **affichage :** **3**
4. lecture de + , c'est un opérateur et le sommet de pile est une (, on l'empile
5. lecture de 4 , c'est un opérande, on l'affiche **affichage :** **34**
6. lecture de) , c'est une parenthèse fermante, on dépile et on affiche les opérateurs dépilés jusqu'à trouver une parenthèse ouvrante qui ne sera pas affichée. Ici la pile sera alors vide.
 affichage : **34+**
7. lecture de * , c'est un opérateur et la pile est vide , on l'empile
8. lecture de (, c'est une parenthèse, on l'empile
9. lecture de 5 , c'est un opérande, on l'affiche **affichage :** **34+5**
10. lecture de - , c'est un opérateur, le sommet de pile étant (, on l'empile
11. lecture de 2 , c'est un opérande, on l'affiche **affichage :** **34+52**
12. lecture de) , c'est une parenthèse fermante, on dépile et on affiche les opérateurs dépilés jusqu'à trouver une parenthèse ouvrante qui ne sera pas affichée. Ici la pile ne sera alors pas vide
 affichage : **34+52-**
13. lecture de \0, dépiler totalement et on affiche les opérateurs trouvés
 affichage : **34+52-***

Voici la manière de procéder en lisant l'expression de l'exemple 2 considérée sous forme de chaîne de caractères sans espace "(a+b*c)-(a*d)" :

1. prendre un pile vide puis démarrer la lecture de l'expression à traduire
2. lecture de (, c'est une parenthèse ouvrante, on l'empile
3. lecture de a , c'est un opérande, on l'affiche **affichage :** **a**
4. lecture de + , c'est un opérateur et le sommet de pile est une (, on l'empile

5. lecture de b , c'est un opérande, on l'affiche **affichage :** **ab**
6. lecture de * , c'est un opérateur et le sommet de pile est + qui un opérateur de priorité inférieure. On empile donc *
7. lecture de c , c'est un opérande, on l'affiche **affichage :** **abc**
8. lecture de) , c'est une parenthèse fermante, on dépile et on affiche les opérateurs dépilés jusqu'à trouver une parenthèse ouvrante qui ne sera pas affichée. Ici la pile sera pas vide
affichage : **abc*+**
9. lecture de - , c'est un opérateur et la pile est vide, on l'empile: **abc*+**
10. lecture de (, c'est une parenthèse ouvrante, on l'empile
11. lecture de a , c'est un opérande, on l'affiche **affichage :** **abc*+a**
12. lecture de * , c'est un opérateur et le sommet de pile est (, on l'empile
13. lecture de d , c'est un opérande, on l'affiche **affichage :** **abc*+ad**
14. lecture de) , c'est une parenthèse fermante, on dépile et on affiche les opérateurs dépilés jusqu'à trouver une parenthèse ouvrante qui ne sera pas affichée. Ici la pile sera pas vide
affichage : **abc*+ad***
15. lecture de \0, dépiler totalement et on affiche les opérateurs trouvés
affichage : **abc*+ad*-**

Objectif 8 : mise en oeuvre de la pile dynamique : exercice d'application n°2

La tour de Hanoï.

Plutôt que d'écrire ici des pages d'explications, je me permets de vous rediriger vers la très bonne vidéo d'un collègue de la Sorbonne :

<https://www.youtube.com/watch?v=rOnRbPKvGQg>

Par la méthode de votre choix (récursive ou itérative), veuillez implanter l'algorithme de ce jeu. Vous empilerez des int et utiliserez trois piles. Pour N=4 jetons, la situation de départ que vous afficherez sera la suivante (sans le cadre) :

1		
2		
3		
4	(vide)	(vide)

ETAT INITIAL

Cet affichage correspond à la figure en haut et à gauche de la diapo de la vidéo située à 4mn59s.

L'affichage suivant que vous produirez sera donc :

2		
3		
4	1	(vide)

Ainsi, vous afficherez et compterez toutes les étapes menant à la solution pour N=4 jetons.

Vous testerez alors pour des valeurs de N supérieures.

Avec 3 disques, voici un exemple de ce qui est attendu de votre part :

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help
>>>>SITUATION DE DEPART :
  1 | | 
  2 | | 
  3 | (vide) (vide)
-----
  2 | | 
  3 | (vide) 1
-----
  3 | 2 | 1
-----
  3 | 1 | 
    | 2 | (vide)
-----
  (vide) | 1 | 
          | 2 | 3
-----
  1 | 2 | 3
-----
  1 | (vide) | 2
          | 3 | 
-----
  (vide) | (vide) | 1
          | 2 | 
          | 3 | 
-----
***** FINI en 7 coups *****
```

```

kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help
>>>>SITUATION DE DEPART :
1 | | 
2 | | 
3 | | 
4 | (vide) | (vide)
-----
2 | | 
3 | | 
4 | 1 | (vide)
-----
3 | | 
4 | 1 | 2
-----
3 | | 
4 | (vide) | 1
2
-----
4 | | 
3 | 1
2
-----
1 | | 
4 | 3 | 2
-----
1 | | 
4 | 2
3 | (vide)
-----

```

```
File Edit View Search Terminal Help
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2

      (vide) |      2 |      1
              |      3 |      4
-----

      |      |      1
      |      |      4
      2 |      3 |
-----

      |      |      1
      |      |      2
      1 |      3 |      4
-----

      |      |      3
      |      |      4
      1 |      (vide) |
-----

      |      |      3
      |      |      4
      2 |      1 |
-----

      |      |      2
      |      |      3
      (vide) |      1 |      4
-----

      |      |      1
      |      |      2
      (vide) |      (vide) |      3
              |      4
-----
***** FINI en 15 coups *****
```

Avec 5 disques, voici ce que vous devez obtenir :

...

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help
-----
      3      |      2      |      1
              |      4      |      5
              |      5      |
-----
      3      |      1      |      4
              |      2      |      5
              |      5      |
-----
    (vide)   |      1      |      3
              |      2      |      4
              |      5      |
-----
      1      |      2      |      3
              |      4      |      5
              |      5      |
-----
      1      |      (vide)  |      2
              |      5      |      3
              |      5      |      4
              |      5      |
-----
    (vide)   |      (vide)  |      1
              |      5      |      2
              |      5      |      3
              |      5      |      4
              |      5      |
-----
***** FINI en 31 coups *****
```

Avec 10 disques, voici ce que vous devez obtenir :

...

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp1sdd2
File Edit View Search Terminal Help

  1      |      |      8
  2      |      |      9
          | (vide) |     10
-----
          |      |      3
          |      |      4
          |      |      5
          |      |      6
          |      |      7
          |      |      8
          |      |      9
  2      |      1  |     10
-----
          |      |      2
          |      |      3
          |      |      4
          |      |      5
          |      |      6
          |      |      7
          |      |      8
          |      |      9
 (vide)   |      1  |     10
-----
          |      |      1
          |      |      2
          |      |      3
          |      |      4
          |      |      5
          |      |      6
          |      |      7
          |      |      8
          |      |      9
 (vide)   | (vide) |     10
-----
***** FINI en 1023 coups *****
```

Séance 3

TP2 Les listes chaînées

Introduction :

Qu'elle soit statique ou dynamique, la pile nécessite une zone mémoire contigue. Dans le cas d'un grand nombre de données à empiler, cet espace mémoire peut être difficile voir impossible à trouver.

Un morcellement de cet espace peut être la solution.

La notion de liste chaînée permet de répondre favorablement à ce problème. En effet chaque maille de la liste est allouée seule et le besoin mémoire est donc plus accessible. Les N données que nous stockions dans une pile demanderont N allocations de mailles mais ces N emplacements mémoire seront dispersés. La contre partie est de devoir mémoriser dans chacune des mailles, l'adresse de la maille suivante.

Le but de ce TP est d'écrire un ensemble liste.h/liste.c contenant des fonctions qui seront précisées plus loin.

Des exercices d'application suivront pour conclure ce TP.

Ce TP se déroulera toujours avec une structure tp2 avec main / liste.h&.c / es.h&.c

Objectif 1 : déclaration d'une liste chaînée, mise en place des fichiers d'entête

1. Placez la déclaration suivante dans votre fichier liste.h :

```
typedef struct maille {  
    T_Elt elt; // type T_Elt toujours déclaré dans es.h  
    struct maille *suivant; //ici se trouve l'auto référence  
    // struct maille *precedent; //à mettre en oeuvre plus tard  
}T_Maille ;
```

2. Dans votre main, déclarez une variable par :

T_Maille *maListe=NULL;

3. Compilez et vérifiez que tout va bien
4. Ecrivez une fonction qui crée une maille et renvoie son adresse. Le prototype sera le suivant :

T_Maille *newMaille(T_Elt, T_Maille *);

Cette fonction alloue un espace mémoire pour une nouvelle maille, y stocke l'élément dans le champ elt et affecte l'adresse reçue dans le champ suivant. Cette fonction renvoie l'adresse de la nouvelle maille allouée.

Objectif 2 : fonction afficherListe

1. Ecrivez une fonction d'affichage itérative dont le prototype sera :
int afficherListe(T_Maille *);
2. L'entier renvoyé permettra de savoir si la liste est vide ou pas. Vous pouvez écrire une fonction **int listeVide(T_Maille *);** ou faire usage d'une macro fonction
#define listeVide(L) L==NULL
3. Ecrivez maintenant une autre fonction d'affichage mais en récursif avec ce prototype :
int afficherListeRec(T_Maille *);
4. Vous testerez les deux fonctions dans l'objectif suivant

Objectif 3 : fonction insererEnTete

Cette fonction insère un élément dans une nouvelle maille qui devient automatiquement la première maille de la liste.

Si la liste était vide lors de cette insertion alors maListe reçoit l'adresse de cette maille et cette maille contient NULL dans son champ suivant.

Si la liste n'était pas vide lors de cette insertion alors le champ suivant de cette maille reçoit la valeur de maListe puis maListe reçoit l'adresse de cette maille. L'ex première maille est désormais la deuxième maille.

1. Dans liste.h , insérez le prototype suivant puis écrire son code dans liste.c :

T_Maille *insererEnTete(T_Elt *,T_Maille *);

On fera appel à la fonction newMaille.

2. insérez plusieurs éléments dans votre liste et vérifiez que tout va bien en affichant régulièrement la liste à l'aide de la fonction vue à l'objectif 2.
3. Dans liste.h , insérez une autre fonction d'insertion en tête puis écrire son code dans liste.c :

void insererEnTete2(T_Elt *,T_Maille **);

4. insérez plusieurs éléments dans votre liste et vérifiez que tout va bien en affichant régulièrement la liste à l'aide de la fonction vue à l'objectif 2.

Objectif 4 : fonction insererEnFin

Cette fonction insère un élément dans une nouvelle maille qui devient automatiquement la dernière maille de la liste.

Si la liste était vide lors de cette insertion alors maListe reçoit l'adresse de cette maille et cette maille contient NULL dans son champ suivant. Cette maille est à la fois la première et la dernière. On pourra faire appel à la fonction **insérerTete** pour gérer ce cas.

Si la liste n'était pas vide lors de cette insertion alors il faut parcourir complètement la liste jusqu'à détection de la maille contenant NULL dans son champ suivant.

Une fois cette maille identifiée, il faut allouer une nouvelle maille et écraser ce NULL par l'adresse de cette maille. N'oubliez pas de ranger à nouveau NULL dans le champ suivant de la nouvelle maille fraîchement allouée.

- 1) Dans liste.h , insérez le prototype suivant puis écrire son code dans liste.c :

T_Maille *insererEnFin(T_Elt *,T_Maille *);

- 2) insérez plusieurs éléments dans votre liste et vérifiez que tout va bien en affichant régulièrement la liste à l'aide de la fonction vue à l'objectif 2.
- 3) Dans liste.h, écrivez maintenant une autre fonction d'insertion en fin mais en récursif avec ce prototype :

T_Maille *insererEnFinRec(T_Elt *,T_Maille *);

- 4) insérez plusieurs éléments dans votre liste et vérifiez que tout va bien en affichant régulièrement la liste à l'aide de la fonction vue à l'objectif 2.

Objectif 5 : fini les doublons avec la fonction appartient

Nous souhaitons désormais construire des listes chaînées d'éléments uniques. Il nous faut donc une fonction d'analyse qui rendra son verdict quant à l'insertion.

1. Dans liste.h , insérez le prototype suivant puis écrire son code dans liste.c :

int appartient(T_Elt , T_Maille *);

Cette fonction renvoie 1 si l'élément appartient déjà à la liste (0 sinon).

2. Modifiez les codes d'insertion précédents afin de garantir l'unicité des éléments dans la liste.
3. Compilez, testez
4. Dans liste.h, écrivez maintenant une autre fonction d'appartenance mais en récursif avec ce prototype :

int appartientRec(T_Elt , T_Maille *);

5. Compilez, testez

Objectif 6 : liste chaînée croissante : fonction `insérerAvecOrdre`

Laissons temporairement de côté les fonctions `insérerEnTete` et `insérerEnFin`.

Nous souhaitons désormais construire des listes chaînées d'éléments uniques et ordonnés (croissants par exemple).

1. Dans `liste.h` , insérez le prototype suivant puis écrire son code dans `liste.c` :

`T_Maille *insérerAvecOrdre(T_Elt , T_Maille *)`;

La fonction appartient sera sollicitée avant l'insertion.

Pour cette fonction, nous devons distinguer quatre cas :

- I. Si la liste est vide alors l'élément est insérer en tête. La fonction renverra l'adresse de cette nouvelle maille unique.
- II. Si la liste n'est pas vide ET si l'élément à insérer est plus petit que l'élément de tête de liste alors, il faut insérer en tête ce nouvel élément. La fonction renverra l'adresse de cette nouvelle maille.
- III. Si la liste n'est pas vide ET si l'élément à insérer est plus grand que le dernier élément de la liste alors, il faut insérer en fin de liste ce nouvel élément. La fonction renverra dans ce cas, l'adresse de la teête de liste (qui ne change pas dans ce cas).
- IV. Si la liste n'est pas vide ET si l'élément à insérer n'est pas dans les deux cas précédent, il faut alors l'insérer au sein d'une nouvelle maille qui se situera entre deux mailles existantes que nous nommerons `mailleGauche` et `mailleDroite`.
Le champ suivant de `mailleGauche` recevra l'adresse de cette nouvelle maille contenant l'élément inséré.
Le champ suivant de cette nouvelle maille recevra l'adresse de `mailleDroite`.

2. Compilez et testez

Objectif 7 : fonction `supprimerElement`

L'objectif est cette fois de faire disparaître un élément de la liste.

1. Dans `liste.h` , insérez le prototype suivant puis écrire son code dans `liste.c` :

`T_Maille *supprimerElement(T_Elt , T_Maille *)`;

La fonction appartient sera sollicitée avant l'insertion.

Pour cette fonction, nous devons distinguer quatre cas :

- I. Si la liste est vide alors il n'y a rien à supprimer. La fonction renverra l'adresse reçue (c'est à dire `NULL`).
- II. Si la liste n'est pas vide ET si l'élément à insérer est plus petit que l'élément de tête de liste alors, il faut insérer en tête ce nouvel élément. La fonction renverra l'adresse de cette nouvelle maille.

- III. Si la liste n'est pas vide ET si l'élément à insérer est plus grand que le dernier élément de la liste alors, il faut insérer en fin de liste ce nouvel élément. La fonction renverra dans ce cas, l'adresse de la tête de liste (qui ne change pas dans ce cas).
- Si la liste n'est pas vide ET si l'élément à insérer n'est pas dans les deux cas précédent, il faut alors l'in

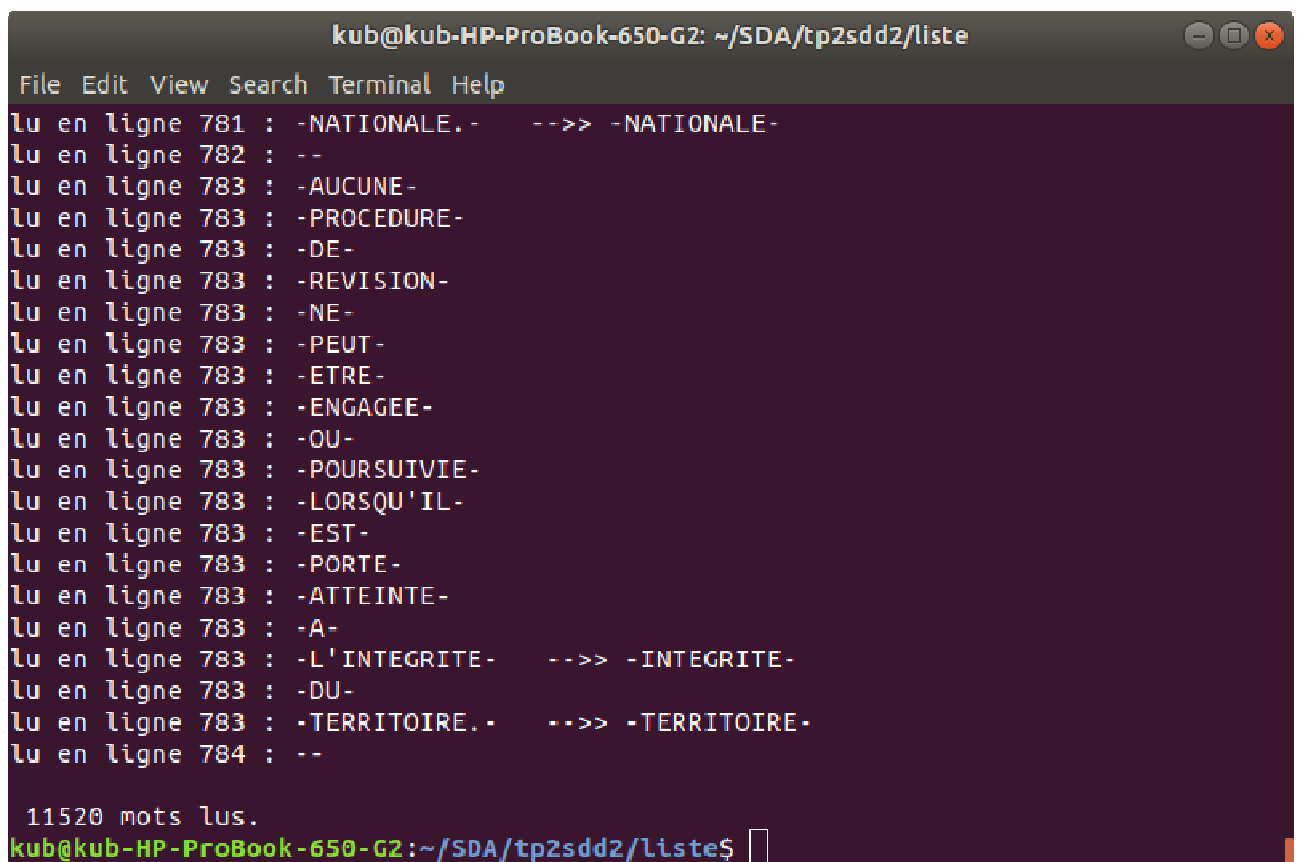
Objectif 8 : fonction tailleListe

Ecrivez une fonction permettant de connaître la taille de la liste (son ombre de mailles). Le prototype sera le suivant :

```
int tailleListe(T_Maille *);
```

Objectif 9 : mise en oeuvre des liste chaînées : exercice d'application n°1

Vous trouverez sur moodle, un fichier nommé **constitution.txt** ainsi qu'un programme en C qui lit ce fichier texte et en affiche chacun des mots à l'écran.



```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp2sdd2/liste
File Edit View Search Terminal Help
lu en ligne 781 : -NATIONALE.- -->> -NATIONALE-
lu en ligne 782 : --
lu en ligne 783 : -AUCUNE-
lu en ligne 783 : -PROCEDURE-
lu en ligne 783 : -DE-
lu en ligne 783 : -REVISION-
lu en ligne 783 : -NE-
lu en ligne 783 : -PEUT-
lu en ligne 783 : -ETRE-
lu en ligne 783 : -ENGAGEE-
lu en ligne 783 : -OU-
lu en ligne 783 : -POURSUIVIE-
lu en ligne 783 : -LORSQU'IL-
lu en ligne 783 : -EST-
lu en ligne 783 : -PORTE-
lu en ligne 783 : -ATTEINTE-
lu en ligne 783 : -A-
lu en ligne 783 : -L'INTEGRITE- -->> -INTEGRITE-
lu en ligne 783 : -DU-
lu en ligne 783 : -TERRITOIRE.- -->> -TERRITOIRE-
lu en ligne 784 : --

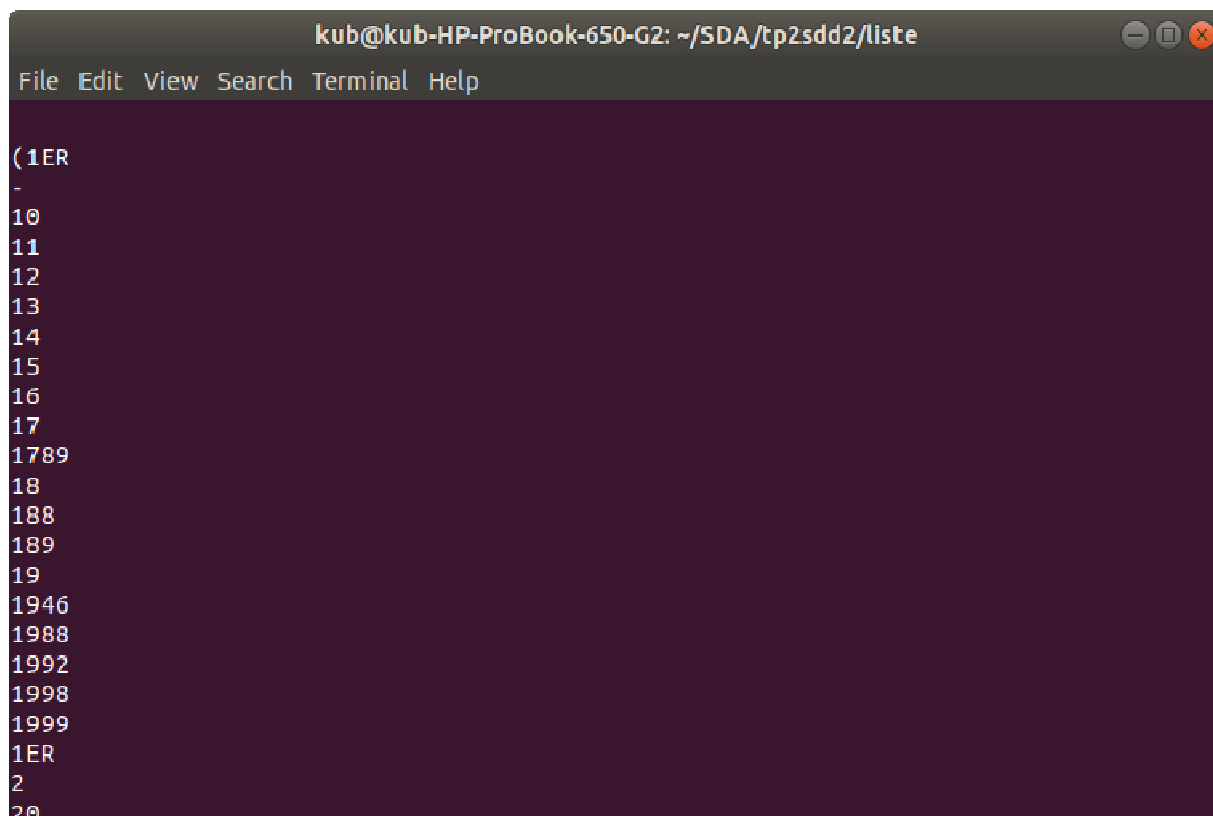
11520 mots lus.
kub@kub-HP-ProBook-650-G2:~/SDA/tp2sdd2/liste$
```

En utilisant les fonctions réalisées précédemment, veuillez modifier le code de ce programme afin de construire une liste chaînée ordonnée des mots lus. Chaque mot sera préalablement passé en majuscule avant insertion dans cette liste.

Vous ferez un affichage de ces mots ainsi triés et indiquerez la taille de la liste chaînée formée.

A ce stade, tous les mots présentant plusieurs occurrences dans le texte initial ne sont présents qu'une seule fois dans la liste chaînée. Nous allons remédier à ce manque dans l'objectif suivant.

Affichage de la liste chaînée obtenue :



```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp2sdd2/liste
File Edit View Search Terminal Help

(1ER
-
10
11
12
13
14
15
16
17
1789
18
188
189
19
1946
1988
1992
1998
1999
1ER
2
20
début de la liste
```

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp2sdd2/liste
File Edit View Search Terminal Help
VOTER
VOTES
VUE
WALLIS
X
XI
XII
XIII
XIV
XV
XVI
Y
[CET
«
»
À
ÉCONOMIQUE
ÉGALITE
ÉTAT
ÉTATS
ÎLES
œUVRE
==>taille de la liste chainee :1673 mailles
kub@kub-HP-ProBook-650-G2:~/SDA/tp2sdd2/liste$
```

fin de la liste chaînée

Objectif 10 : mise en oeuvre des liste chaînées : exercice d'application n°2

Nous allons enrichir notre travail en mémorisant dans nos données, la ou les ligne(s) où se trouve chaque mot.

Nous allons nous reposer sur notre liste chaînée ainsi constituée.

Cependant, dans chaque maille de la liste, nous remplacerons la chaîne de caractères précédemment stockée par une structure elle-même composée de cette chaîne de caractères et d'une pile dynamique mémorisant le ou les numéros de ligne où la chaîne de caractères se trouve dans la constitution.

1. renommez vos fichiers es.h et es.c en esL.h et esL.c
2. dans ces 2 fichiers, renommez le type T_Elt en T_EltL et ajustez le code en conséquence
3. copiez dans votre répertoire de travail les fichiers es.h es.c pile.h et pile.c du TP1
4. renommez vos fichiers es.h et es.c en esP.h et esP.c. A ce stade, vous pouvez aussi renommez le type T_Elt en T_EltP (facultatif)
- 5.

```
kub@kub-HP-ProBook-650-G2: ~/SDA/tp2sdd2/liste
File Edit View Search Terminal Help
-->URGENCE: 495<-385<-bas de pile
-->UTILE: 385<-bas de pile
-->V: 248<-bas de pile
-->VACANCE: 197<-73<-61<-61<-59<-bas de pile
-->VALABLE: 453<-bas de pile
-->VEILLE: 612<-488<-481<-41<-bas de pile
-->VERS: 664<-bas de pile
-->VERTU: 742<-458<-449<-333<-306<-170<-4<-bas de pile
-->VI: 441<-bas de pile
-->VIE: 473<-121<-37<-bas de pile
-->VIGUEUR: 705<-703<-703<-696<-595<-383<-314<-306<-bas de pile
-->VII: 468<-bas de pile
-->VIII: 513<-bas de pile
-->VINGT: 392<-220<-106<-61<-57<-bas de pile
-->VINGT-QUATRE: 421<-bas de pile
-->VIOLATION: 767<-bas de pile
-->VISE: 708<-614<-bas de pile
-->VISEES: 703<-408<-bas de pile
-->VISES: 537<-535<-bas de pile
-->VOCATION: 626<-bas de pile
-->VOIE: 639<-602<-24<-bas de pile
-->VOIX: 475<-bas de pile
-->VOLONTE: 134<-4<-bas de pile
-->VOTE: 778<-772<-760<-748<-570<-429<-421<-419<-390<-379<-370<-365<-356<-2
93<-215<-213<-184<-149<-bas de pile
-->VOTEE: 748<-421<-bas de pile
-->VOTEES: 374<-368<-bas de pile
-->VOTER: 285<-bas de pile
-->VOTES: 570<-419<-386<-202<-121<-bas de pile
-->VUE: 359<-67<-4<-bas de pile
-->WALLIS: 657<-bas de pile
-->X: 573<-bas de pile
-->XI: 609<-595<-bas de pile
-->XII: 621<-bas de pile
-->XIII: 712<-660<-bas de pile
-->XIV: 731<-bas de pile
-->XV: 739<-488<-bas de pile
-->XVI: 773<-bas de pile
-->Y: 730<-708<-688<-671<-602<-495<-414<-361<-327<-89<-53<-4<-bas de pile
-->[CET: 762<-bas de pile
-->«: 19<-17<-bas de pile
-->»: 20<-18<-bas de pile
-->À: 769<-769<-bas de pile
-->ÉCONOMIQUE: 595<-bas de pile
-->ÉGALITE: 19<-bas de pile
-->ÉTAT: 758<-723<-718<-705<-703<-698<-695<-689<-650<-634<-612<-539<-529<-527<-5
27<-500<-449<-327<-321<-314<-306<-279<-274<-262<-117<-117<-116<-42<-bas de pile
-->ÉTATS: 742<-737<-734<-456<-bas de pile
-->ÎLES: 657<-bas de pile
-->œUVRE: 721<-35<-bas de pile
-->>taille de la liste chainee :1672 mailles
kub@kub-HP-ProBook-650-G2:~/SDA/tp2sdd2/liste$
```

Bonus pour les plus rapides

Vous avez terminé les deux sujets précédents.

Nous vous proposons de modéliser, coder et tester le concept suivant.

à écrire, à compléter ...

Bibliographie

Pointeurs, allocation dynamique de mémoire :

<https://docs.google.com/presentation/d/1Q2Q1UYuTEE2jIY9kkne-HAtLekYTEZm5NEjVbCHrKWI/edit?usp=sharing>

Types abstraits de données :

<https://docs.google.com/presentation/d/1RWRCT8R464MDsOCAa4qDq0fsSjJoRYHzlk1dcu9xTDQ/edit?usp=sharing>