SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 1
Pointeurs des chaînes

Version du 16 septembre 2020

Objectif(s)

- ✯ Rappels sur la gestion de la mémoire.
- ✯ Pointeurs, arithmétique sur pointeurs.
- ✯ Allocation dynamique.
- ✯ Chaînes de caractères.

TD : Des pointeurs aux chaînes

Exercice 1 *(base)* – Manipulations simples de pointeurs

1. On souhaite écrire une fonction qui calcule le minimum, le maximum et la moyenne d'un tableau d'entiers. Une fonction n'a qu'une seule valeur de retour, comment faire pour renvoyer ces trois valeurs ? Donner le prototype de la fonction nommée `min_max_moy`.
2. Ecrivez le corps de la fonction ainsi qu'un `main` pour tester votre fonction.
3. Écrivez une version récursive de cette fonction. Pour cela vous écrirez une fonction intermédiaire permettant de calculer par récursion le minimum, le maximum et la somme des éléments d'un tableau. La récursion se fera en appelant récursivement la fonction privée de son premier élément jusqu'à atteindre le cas trivial d'un tableau ne contenant qu'une seule valeur.
4. On souhaite tester cette fonction sur plusieurs tableaux, notamment sur des tableaux de taille variée. Ecrivez un programme testant cette fonction sur des tableaux de taille 1 à 100 (vous utiliserez pour cela une allocation dynamique de la mémoire). Un tableau de taille `n` contiendra les `n` premiers entiers (de cette façon, le minimum, le maximum et la moyenne seront connus et pourront être vérifiés). Peut-on faire cela avec une boucle de façon à ne pas répéter le même code 100 fois ? Comment faire ? Ecrivez la fonction `main` correspondante en vérifiant à chaque fois que les minimum, maximum et moyenne sont corrects.

Exercice 2 *(obligatoire)* – Décomposition d'une chaine de caractères en mots

Tout traitement de texte standard a besoin de distinguer les mots composant un paragraphe ne serait-ce que pour réaliser, par exemple, la justification de ce paragraphe. L'objet de cet exercice va être de décomposer une chaîne de caractères quelconque contenant des mots séparés par des espaces en un tableau de pointeurs sur des mots. Soit la chaîne suivante :

```
mot1␣et␣mot2␣et␣mot3
```

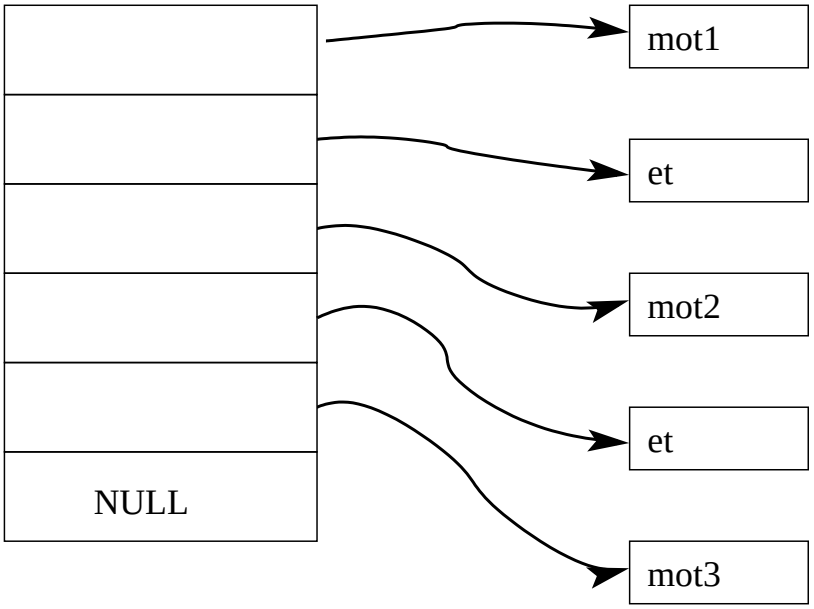



FIGURE 1 – Tableau de pointeurs sur les mots de la chaîne de caractères.

On veut décomposer cette chaîne en un tableau de pointeurs tel que représenté sur la Figure 1.

Comme vous pouvez le voir sur la figure, le tableau comporte toujours une case de plus que le nombre de mots de la chaîne. Le nombre de mots de la chaîne n'étant disponible nulle part ailleurs, la case suivant la dernière case remplie contient toujours un pointeur égal à `NULL` et joue le rôle de marqueur de fin. Il s'agit d'un cas particulier pour cet exercice et non d'une généralité en C (contrairement à la présence d'un caractère '
0' à la fin d'un tableau de caractères, qui est elle obligatoire pour les chaînes de caractères).

Cet énoncé vous propose une série d'exercices basés sur la manipulation de cette structure de données dans l'objectif de vous familiariser avec l'allocation dynamique et l'utilisation massive de pointeurs. La construction de cette structure n'étant pas l'étape la plus simple nous allons commencer par quelques exercices permettant de manipuler cette structure.

Fonction manipulant **une chaine de caractères**

1. Écrivez la commande permettant d'allouer la chaîne de caractères d'un mot constitué de `nb_char` caractères comme celle fournie en exemple précédemment (`mot1_et_mot2_et_mot3`).
2. Écrivez une fonction permettant de compter le nombre de mots d'une chaîne de caractères. Prototype :
   ```
   int compte_mots_chaine(char *chaine);
   ```

Fonctions manipulant le **tableau de chaines de caractères**

3. Écrivez le prototype d'une fonction `compte_mots` qui parcourt le tableau et retourne le nombre de mots.
4. Écrivez cette fonction.

   Nous entrons maintenant dans le coeur du sujet, l'allocation et la destruction (libération mémoire) de cette structure de données. Nous allons commencer par la destruction et pour cela, nous prendrons en compte le fait que aussi bien le tableau de pointeurs que les chaînes de caractères ont été alloués dynamiquement en faisant appel à la fonction `malloc`.
5. Écrivez la fonction :
   ```
   void detruit_tab_mots(char **ptab_mots)
   ```

   Cette fonction libérera l'intégralité de la mémoire allouée en utilisant la fonction `free`.

Fonctions permettant de passer d'une **chaine de caractères** au **tableau de chaines de caractères**.

Vient maintenant la construction de la structure de données à partir d'une chaîne de caractères contenant des mots séparés par des espaces. Cette chaîne source passée en argument ne devra en aucun cas être modifiée, et aucune hypothèse ne devra être faite concernant sa pérennité.

6. Écrivez la commande permettant d'allouer le tableau en supposant que celui-ci doive contenir `nb_mots` mots et donc `nb_mots + 1` éléments.
7. Écrivez la fonction :

```
char **decompose_chaine(char *chaine);
```

Cette fonction parcourt la chaîne `chaine` fournie en argument, alloue le tableau, alloue les chaînes correspondant aux mots et retourne un pointeur sur le tableau alloué. Si aucun mot n'est trouvé dans la chaîne, la fonction retourne la valeur `NULL`.

TME : Chaînes et tableaux de chaînes

Vous devrez écrire vos fonctions dans le fichier `csvl.c`.

Si besoin et à de rares exceptions près, les fonctions vues en TD vous seront fournies. Pour cette première séance et pour faire un démarrage en douceur, nous vous demanderons cependant d'implémenter les fonctions de comptages vues en TD.

Exercice 3 *(obligatoire)* – Décomposition d'une chaine de caractères en mots (suite)

1. Écrivez la fonction de comptage de mots dans une chaîne de caractères (`compte_mots_chaine`) vue en TD et écrivez une fonction main permettant de la tester. D'une manière générale, il est important que vous testiez chaque fonction écrite dès que possible. Ces tests ne seront pas forcément conservés dans le programme final, mais ils permettent de détecter les erreurs au plus tôt.
2. De même que précédemment, écrivez la fonction de comptage des mots figurant dans un tableau de mots et complétez votre fonction main pour la tester. Pour cela, ajouter à votre programme la déclaration du tableau contenant les chaines "mot1", "et", "mot2", "et", "mot3" et le test de la fonction écrite précédemment.

   ATTENTION : Si un tableau statique à une dimension est équivalent à un pointeur, ce n'est pas le cas pour un tableau à deux dimensions. Vous devez donc déclarer votre tableau sous la forme d'un tableau de pointeurs. Pour l'initialiser, vous pouvez utiliser le fait que lorsque vous écrivez "mot1", cela correspond à une chaîne de caractères constante stockée en mémoire. Vous pouvez donc tout à fait écrire :

```
char *s="mot1";
```

   Dans ce cas, `s` pointera sur la zone mémoire constante (allouée statiquement).
3. Écrivez la fonction d'affichage des mots stockés dans un tableau de mots et complétez votre main pour tester la fonction.
4. Écrivez maintenant l'opération de recomposition de la chaîne initiale à partir du tableau de mots. Son prototype est le suivant :

```
char *compose_chaine(char **ptab_mots)
```

   Cette fonction alloue un espace mémoire suffisant pour stocker la chaîne reconstituée et copie les mots un à un dans cet espace. Elle retourne ensuite son adresse mémoire. Elle ne modifie pas le tableau de mots fournis en argument.

   Note : la chaîne recomposée sera généralement identique à la chaîne initiale, sauf dans le cas où celle-ci contenait plusieurs espaces successifs, qui dans la chaîne recomposée seront remplacés pas des espaces uniques.

5. Mettez à jour votre fonction de test pour vérifier le fonctionnement de la fonction de la question précédente en n'oubliant pas de libérer la mémoire allouée.

Exercice 4 *(approfondissement)* – Compactage de la structure mémoire

Il est très fréquent que dans un texte un mot soit présent plusieurs fois, c'est le cas du mot "et" dans la chaîne présentée en exemple. Tel que nous avons réalisé la construction de la structure il sera présent deux fois en mémoire (à des adresses différentes). Bien que maintenant les quantités de mémoire puissent paraître sans limites, il s'agit d'un gaspillage d'espace. Pour éviter ce gaspillage nous allons modifier notre structure de données pour garantir l'unicité en mémoire de chaque mot. Dans l'exemple vu précédemment, cela donnera le tableau représenté sur la Figure 2.

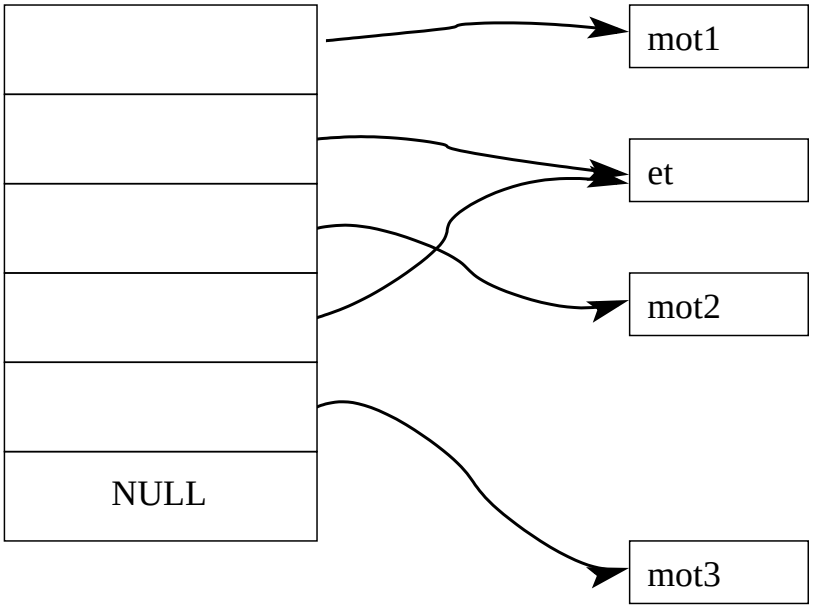


FIGURE 2 – Tableau de chaînes "compacté".

Maintenant si un mot est présent deux fois, les deux cases du tableau pointeront sur la même chaîne de caractères.

1. Écrivez la fonction :

   ```
   char **reduit_tab_mots(char **ptab_mots)
   ```

   Cette fonction parcourt le tableau, supprime les occurrences multiples d'un mot, met à jour les cases du tableau correspondantes et retourne un pointeur sur le tableau.

2. Le compactage a-t-il un impact sur la fonction de destruction de la mémoire, si oui comment résoudre le problème ?

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 2
Listes chaînées et simulation d'un écosystème - partie 1

Version du 16 septembre 2020

Objectif(s)

- ✯ Les structures sont supposées acquises
- ✯ rand et srand (rappels)
- ✯ Listes chaînées (rappels)
- ✯ assert
- ✯ Makefile
- ✯ Bonnes pratiques de programmation : les tests
- ✯ Bonnes pratiques de programmation : ddd/gdb et valgrind

L'objectif de ce sujet est la réalisation d'un petit projet de programmation d'un écosystème virtuel utilisant des listes chaînées. Nous allons d'abord commencer par un petit exercice de rappels sur les listes puis aborderons le problème de l'écosystème.

TD Rappels sur les listes chaînées

L'objectif de cette première partie est de prendre en main le concept de listes chaînées au travers de quelques exercices pédagogiques.

Soit la structure de données suivante :

```
typedef struct _elt Elt;
struct _elt{
  int donnee;
  Elt *suivant;
};
```

Exercice 1 *(base)* – Rappels sur les listes, analyse d'un exemple

Soit le programme suivant :

```
int main() {
  int taille=10;
  Elt *liste=NULL;
  Elt *nelt=NULL;
  int i=0;

  for (i=0;i<taille;i++) {
    nelt=malloc(sizeof(Elt));
```

```
    if (nelt == NULL) {
      printf("Erreur lors de l'allocation.\n");
      return 0;
    }
    nelt->donnee=i;
    nelt->suivant=liste;
    liste=nelt;
  }

  nelt=liste;
  while (nelt) {
    printf("%d ",nelt->donnee);
    nelt=nelt->suivant;
  }
  printf("\n");

  return 0;
}
```

1. Que fait ce programme ? Qu'est-ce qui est affiché à l'écran ?
2. Quelle est la place mémoire occupée par la liste chaînée créée dans ce programme ? Quelle taille ferait un tableau contenant les mêmes données ?
3. La mémoire allouée pour cette liste n'a pas été libérée. Ajoutez des instructions permettant de libérer toute la mémoire qui a été allouée.

TD Ecosystème - Mise en place

L'objet de cette partie du sujet est d'écrire des fonctions de manipulation de listes chaînées et de les utiliser pour programmer une simulation simple d'écosystème.

Le modèle d'écosystème que vous allez programmer n'a aucune prétention à être réaliste, mais il permet de se familiariser avec le concept d'équilibre, primordial dans un écosystème. Cet écosystème contiendra deux types d'entités virtuelles : des proies et des prédateurs, susceptibles de manger ces dernières. Notre écosystème est un monde discret (un tore, que nous afficherons comme un rectangle) contenant un certain nombre de cases, identifiées par leurs coordonnées (entières) x et y. Chaque proie (et chaque prédateur) est dans une case donnée et peut se déplacer. A un instant donné, une case peut contenir plusieurs proies et plusieurs prédateurs. Chaque case peut aussi contenir de l'herbe, la nourriture des proies.

La simulation de notre écosystème repose sur plusieurs structures de données et sur des fonctions que vous allez écrire dans la suite de cette séance. Les données utilisées pour la simulation sont une liste chaînée contenant les proies et une autre liste chaînée contenant les prédateurs.

Exercice 2 *(obligatoire)* – Organisation du programme et compilation séparée

Le programme est organisé en quatre fichiers :
- `main_tests.c`, qui contiendra un main avec les tests de vos fonctions,
- `main_ecosys.c`, qui contiendra un main permettant de simuler un écosystème,
- `ecosys.c` qui contient toutes les autres fonctions de manipulation de listes.
- `ecosys.h` qui contient les prototypes des fonctions de ecosys.c et les définitions de structure (.h).

1. A votre avis, dans quels fichiers .c le fichier `ecosys.h` sera-t-il inclus ?
2. Donnez les lignes de commande permettant de compiler séparément chaque fichier .c puis de créer l'exécutable `test_ecosys` à partir de `main_test.o` et `ecosys.o` et l'exécutable `ecosys` à partir de `main_ecosys.o` et `ecosys.o`
3. Le fichier `main_test.c` a été modifié. Quelles commandes sont nécessaires pour que les exécutables soient à jour ?

4. Le fichier `ecosys.c` a été modifié. Quelle commandes sont nécessaires pour que les exécutables soient à jour ?
5. Ces opérations fastidieuses de recompilation sont grandement facilitées par l'utilisation de l'utilitaire make. Ecrivez le `Makefile` permettant de compiler les programmes `tests_ecosys` et `ecosys`.

Exercice 3 *(obligatoire)* – Structure de données

Les proies et les prédateurs seront représentés par une même structure de données contenant les coordonnées entières `x` et `y`, un nombre réel `energie` (tel qu'un animal dont l'énergie tombe en dessous de 0 est mort, ce point sera géré plus tard). Il contiendra ensuite un tableau `dir` de deux entiers qui représentera sa direction (nous y reviendrons également plus tard). Comme nous souhaitons stocker des variables de ce type dans des listes simplement chaînées, la structure contiendra enfin un pointeur nommé `suivant` sur cette même structure.

Ces déclarations seront réalisées dans `ecosys.h`, de même que les prototypes des fonctions que vous écrirez par la suite. Les fonctions elles-mêmes seront écrites dans un fichier `ecosys.c`.

1. Ecrivez la déclaration de cette structure, à laquelle vous donnerez, via un `typedef` le nom équivalent `Animal`.
2. Ecrivez la fonction de création d'un élément de type `Animal` (allocation dynamique). Vous initialiserez les champs `x`, `y`, et `energie` à partir des arguments de la fonction. Les cases du tableau `dir` seront initialisées aléatoirement avec les valeurs -1, 0 ou 1.

   La fonction aura le prototype suivant :

   ```
   Animal *creer_animal(int x, int y, float energie);
   ```

   Pour que votre programme soit robuste et éviter les erreurs de segmentation lors du développement du programme, nous utiliserons des appels à `assert(expression);` qui (comme en python) arrêtent le programme lorsque l'expression est fausse. Par exemple, dans cette fonction, il faut vérifier que le pointeur retourné par `malloc` n'est pas `NULL`, ce que vous ferrez avec un appel à `assert`.
3. Ecrivez la fonction d'ajout en tête dans la liste chaînée.

   La fonction aura le prototype suivant :

   ```
   Animal *ajouter_en_tete_animal(Animal *liste, Animal *animal);
   ```

   N'oubliez par d'utiliser `assert` pour éviter les erreurs de segmentation potentielles.
4. Ecrivez des fonctions permettant de compter le nombre d'éléments contenus dans une liste chaînée. Vous écrirez une fonction itérative et une fonction récursive.

   Les fonctions auront les prototypes suivants :

   ```
   unsigned int compte_animal_rec(Animal *la);
   unsigned int compte_animal_it(Animal *la);
   ```

Exercice 4 *(obligatoire)* – Affichage et main

1. La fonction d'affichage de votre ecosystème affichera le contenu des différentes cases de votre monde simulé. Vous afficherez un espace pour les cases vides, une étoile ('*') pour les cases contenant au moins une proie, un 'O' pour les cases contenant au moins un prédateur et si une case contient des proies et des prédateurs, vous afficherez un '@'.


```
Nb proies (*):        20
Nb predateurs (O):    20
+----------+
| *        |
|* *O * **O|
| O*O      |
|  O  O    |
|  @  O  O |
|  **    *O|
|     @ O @|
|    *   O@|
|  O *O    |
|  O O @*  |
+----------+
```

Pour cela vous déclarerez un tableau statique de **char** à 2 dimensions de taille `SIZE_X*SIZE_Y` et commencerez par le remplir d'espace. Ensuite, vous parcourrez la liste des proies et mettrez une étoile dans la case où les proies se trouvent, puis ferez de même avec les prédateurs. Enfin, vous afficherez ce tableau de caractères pour obtenir l'affichage précédent.

L'herbe mentionnée en introduction n'est pas représentée visuellement ici. Vous pourrez l'ajouter pendant le TME.

TME : Ecosystème - Simulation

Cette partie est à faire en TME. Les fonctions de la partie précédente, corrigées en TD, sont fournies.

Exercice 5 *(obligatoire)* – Tests des fonctions et main

1. Ecrivez la fonction permettant d'ajouter un animal à la position `x`, `y`. Cet animal sera ajouté à la liste chaînée `liste_animal`. Plutôt que de renvoyer l'adresse du premier élément de la liste, vous passerez ce pointeur par adresse de façon à pouvoir le modifier directement dans la fonction : l'argument sera donc un pointeur sur liste chaînée, donc un pointeur sur un pointeur sur un Animal... Vous vérifierez (avec assert) que les coordonnées données sont correctes c'est-à-dire positives et inférieures à `SIZE_X` ou `SIZE_Y` qui sont des étiquettes (**#define**) définies par ailleurs.

   La fonction aura le prototype suivant :

   ```
   void ajouter_animal(int x, int y, Animal **liste_animal);
   ```

2. Ecrivez la fonction `liberer_liste_animaux(Animal *liste)` qui permet de libérer la mémoire allouée pour la liste `liste`.
3. Ecrivez une fonction main qui va créer quelques proies et quelques prédateurs à des positions variées, vérifiez leur nombre en faisant appel aux fonctions de comptage que vous avez définies et enfin affichez l'état de votre écosystème puis libérez la mémoire allouée pour ces listes. Vous écrirez cette fonction dans le fichier `main_tests.c`. Vous complèterez aussi le `Makefile` pour créer le programme `tests_ecosys`.
4. Pour vérifier qu'il n'y a pas de fuite mémoire, il faut utiliser l'utilitaire valgrind. Lancez votre programme avec lui (en tapant `valgrind` puis votre programme - ex : `valgrind ./tests_ecosys`). L'objectif est de n'avoir aucun octet qui soit perdu :

   ```
   ==66002== LEAK SUMMARY:
   ==66002==    definitely lost: 0 bytes in 0 blocks
   ==66002==    indirectly lost: 0 bytes in 0 blocks
   ```

5. Ecrivez la fonction permettant d'enlever un élément de la liste chaînée et de libérer la mémoire associée. Comme précédemment, la liste sera passée par adresse.

   La fonction aura le prototype suivant :

   ```
   void enlever_animal(Animal **liste, Animal *animal);
   ```

   Ajoutez dans votre main la suppression de quelques proies et quelques prédateurs avant la libération de la liste entière, tout en vérifiant que les comptages sont toujours corrects. Vérifiez à nouveau qu'il n'y a pas de fuite mémoire avec valgrind.

TD/TME Semaine 3
Listes chaînées et simulation d'un écosystème - partie 2

Version du 16 septembre 2020

Objectif(s)

- ✯ Les structures sont supposées acquises
- ✯ tableaux 2D statiques en arguments
- ✯ lecture et ecriture de fichier
- ✯ variables globales
- ✯ Listes chaînées (suite)

TD

Exercice 1 *(obligatoire)* – Tableaux 2D, statiques et dynamiques

1. Soit le `main` suivant

```
#define DIM1 5
#define DIM2 6

int main(void) {

   char tab2D[DIM1][DIM2];

   InitTab(tab2D);

   return 0;
}
```

   La fonction `InitTab` permet de remplir le tableau de 0. Ecrire cette fonction. Attention au prototype...

2. Transformez ce programme en remplaçant le tableau statique par un tableau dynamique. Attention aux fuites mémoires. L'initialisation du tableau dynamique se fera au niveau du `main` et `InitTab` ne fera toujours que le remplissage de 0. Son prototype pourra être modifié.
3. Quels sont les avantages et inconvénients des tableaux statiques et dynamiques ? Dans quels cas utiliser l'un ou l'autre ?

Exercice 2 *(base)* – Variables globales

Considérons le programme suivant écrit dans le fichier `main.c` :

```
#include <stdio.h>

int a, b, c;

int f(int x) {
   return a*x;
}

int g() {
   int c;

   c=a;
   a=b;
   b=c;

   return c;
}

int main() {
   int x,y;

   a=2;
   b=3;
   c=4;
   x=f(a);
   y=g();
   printf("%d %d %d %d %d\n", x, y, a, b, c);

   return 0;
}
```

1. 1. Quelles sont les variables globales ?
   2. Quelles sont les variables locales à f ?
   3. Quelles sont les variables locales à g ?
   4. Quelles sont les variables locales à main ?
   5. Quelles sont les paramètres formels de f ?
   6. Quelles sont les paramètres formels de g ?
   7. Quelles sont les paramètres effectifs de f ?
   8. Que vaut c à la fin de l'appel à la fonction g ?
   9. Que fait la fonction g ?
   10. Qu'affiche le programme ?
   11. Que retourne le programme ?
2. Ce programme est maintenant découpé en 3 fichiers `.c` et autant de `.h` :
   — `main.c` contient la fonction `main`,il a un fichier .h associé,
   — `fonction1.c` contient la fonction `f`, il a un fichier .h associé,
   — `fonction2.c` contient la fonction `g`, il a un fichier .h associé.
   Ecrire le contenu des fichiers `.c` et `.h` (il n'est cependant pas nécessaire de recopier le corps des fonctions).

1 Lecture et écriture de fichier

Exercice 3 *(obligatoire)* – Lecture avec `fscanf`

Voici un fichier contenant les notes d'étudiants :

```
3
567801 Piet Pompies 3 LU1IN018 12  LU2IN019 10  LU2IN011 12
567802 John Doe 4 LU1IN019 16  LU2IN012 -1  LU2IN010 14  LU2IN014 -1
567803 Jan Alleman 2 LU2IN003 -1  LU2IN019 11
```

La première ligne contient le nombre d'étudiants. Ensuite, chaque ligne correspond à un étudiant et contient l'identifiant, le prénom, le nom, le nombre d'ue et pour chaque ue le code et la note (-1 s'il n'a pas de note).

On suppose qu'il n'y a pas plus de 20 UE, que les codes UE sont sur 8 caractères et que les noms et prénoms font moins de 200 caractères. La structure pour un étudiant est :

```
typedef struct _ty_etu{
   int id_etu;
   char nom[200], prenom[200];
   int nb_ue;
   char codes_ue[9][20];
   int notes[20];
} ty_etu;
```

1. Ecrivez la fonction `lecture_ascii_etu` permettant de lire ce fichier et qui retourne le tableau d'étudiants qu'elle aura alloué et rempli, ainsi que le nombre d'étudiants. Cette fonction prendra en argument le nom du fichier à lire.
2. S'il faut lire et écrire rapidement la base de données des étudiants, la lecture et l'écriture binaire sont beaucoup plus rapides. Ecrivez la fonction `ecriture_binaire_etu` permettant d'écrire ces informations au format binaire.
3. Ecrivez maintenant la fonction `lecture_binaire_etu` permettant de lire ces informations au format binaire.

Exercice 4 *(entraînement)* – Fonctions sur les listes

Nous reprenons ici la structure introduite dans l'exercice 1 de la semaine 2 permettant de stocker des entiers dans une liste chaînée.

1. Ecrivez une fonction permettant d'extraire les éléments de la liste contenant un chiffre pair et indiquez comment l'utiliser sur la liste définie dans le programme ci-dessus. Les éléments de la liste contenant un chiffre impair devront être libérés et les chaînages entre les éléments contenant des chiffres pairs seront modifiés si nécessaire. Prototype :

   ```
   Elt *filtre_pair1(Elt *liste);
   ```

2. On souhaite écrire la même fonction mais avec un seul argument et sans valeur de retour. Quel sera le prototype de cette fonction ? Ecrivez-la en vous inspirant de la question précédente et indiquez comment utiliser cette fonction sur la liste définie dans le programme précédent.

TME : Ecosystème - suite

Nous allons cette semaine simuler notre écosytème. Il fonctionne en temps discret. A chaque pas de temps un certain nombre d'opérations devront être réalisées :

- toutes les proies se déplacent, éventuellement changent de direction de déplacement, leur énergie est décrémentée de 1 ;
- si de l'herbe est disponible, une proie regagne `gain_energie_proie` energie ;
- les proies sont susceptibles de se reproduire avec une probabilité `p_reproduce_proie` ;
- tous les prédateurs se déplacent, éventuellement changent de direction de déplacement, leur énergie est décrémentée de 1 ;
- les prédateurs qui sont sur la même case qu'une proie ont une probabilité `p_manger` de les dévorer. Dans ce cas la proie meurt et le prédateur augmente son énergie d'un montant valant l'énergie de la proie ;
- les prédateurs sont susceptibles de se reproduire avec une probabilité `p_reproduce_predateur`.

Les différentes probabilités évoquées ci-dessus seront des variables globales que vous déclarerez dans `ecosys.c`. Vous les déclarerez en tant qu'`extern` dans `ecosys.h`. Voici leur liste, avec des exemples de valeurs possibles :

```
 /* Parametres globaux de l'ecosysteme (externes dans le ecosys.h)*/
float p_ch_dir=0.01; //probabilte de changer de direction de dsplacement
int gain_energie_proie=6;
int gain_energie_predateur=20;
float p_reproduce_proie=0.4;
float p_reproduce_predateur=0.5;
int temps_repousse_herbe=-15;
```

Nous allons à présent détailler les différentes fonctions permettant de programmer cette simulation.

Exercice 5 *(obligatoire)* – Déplacement et reproduction

Les mouvements seront gérés de la façon suivante : chaque proie (ou prédateur) dispose d'une direction indiquée dans le champ `dir`, qui est un tableau de deux entiers. Il indique la direction suivie sous la forme de deux entiers valant -1, 0 ou 1. Ces valeurs indiquent de combien les coordonnées de la proie doivent être décalées. Considérant que la première case de la prairie est en haut à gauche, une direction de $(1, -1)$ correspond, par exemple, à un mouvement vers la case en haut (1) et à droite (-1), une direction de $(0, 0)$ correspond à une proie immobile.
Avant chacun de ses déplacements, chaque animal peut opérer un changement de direction avec une probabilité `p_ch_dir`, autrement dit si un nombre aléatoire compris entre 0 et 1 est inférieur à cette valeur. Pour obtenir ce nombre aléatoire, il faut utiliser la fonction `rand` qui renvoie un entier et le diviser par la valeur maximum, à savoir `RAND_MAX`. Rappel : toutes les "probabilités" seront déclarées sous forme de variables globales dans le fichier contenant la fonction main.

1. Ecrivez une fonction permettant de faire bouger tous les animaux contenus dans la liste chaînée passée en argument. Le déplacement de l'animal se fera dans la direction indiquée par le champs `dir`. Le monde sera supposé torique, c'est-à-dire que si la proie essaie d'aller en haut alors qu'elle est sur la première ligne, elle se retrouvera automatiquement sur la dernière ligne. De même si une proie essaie d'aller à droite alors qu'elle est sur la dernière colonne de droite, elle réapparait sur la même ligne et sur la colonne la plus à gauche.

   La fonction aura le prototype suivant :

   ```
   void bouger_animaux(Animal *la);
   ```

   ATTENTION : il faut bien vérifier que les coordonnées sont correctes, et notamment positives. `x=x%10` ne garantit pas que `x` sera positif ! Une erreur à ce niveau peut provoquer des erreurs de segmentation dans la fonction d'affichage, erreurs de segmentation TRES difficiles à détecter (les debogueurs n'y voient que du feu...).
2. Testez vos fonctions dans le main de tests en ne créeant qu'un animal à une position que vous aurez définie et en le déplaçant dans une direction que vous aurez aussi définie. Vérifiez bien la toricité du monde.
3. Ecrivez une fonction permettant de gérer la reproduction des animaux. Vous parcourrez la liste passée en argument et, pour un animal `ani`, vous ajouterez un nouvel animal à la même position qu'`ani` avec une probabilité `p_reproduce`. Le nouvel animal a pour énergie la moitié de celle de son parent et le parent a son énergie divisée par 2.

   Remarque : l'ajout se fera en tête, un animal qui vient de naître ne sera pas considéré par votre boucle et ne pourra donc pas lui-même se reproduire lors de ce pas de temps. Il pourra cependant, bien sûr, se reproduire lors des pas de temps suivants avec la même probabilité que son parent.

   La fonction aura le prototype suivant :

   ```
   void reproduce(Animal **liste_animal, float p_reproduce);
   ```
4. Testez vos fonctions dans le main de tests mettant le taux de reproduction à 1 et en vérifiant que le nombre d'animaux est bien multiplié par 2.

Exercice 6 *(obligatoire)* – Gestion de l'herbe

Les proies mange de l'herbe, mais la quantité d'herbe est limitée : si une proie mange l'herbe d'une case, il n'y en a plus et l'herbe met un certain temps à repousser. La quantité d'herbe et le temps de repousse sont modélisés par un tableau à 2 dimensions d'entiers. Si la valeur d'une case est positive ou nulle, il y a de l'herbe, sinon non.

1. Dans le fichier `main_ecosys.c` vous déclarerez un tableau statique à 2 dimensions d'entiers de taille `SIZE_X*SIZE_Y` et vous l'initialiserez à 0 dans toutes ses cases.
2. A chaque itération de la simulation, la quantité d'herbe de chaque case est incrémentée de 1. Ecrivez une fonction `void rafraichir_monde(int monde[SIZE_X][SIZE_Y])` qui effectue cette opération.

Exercice 7 *(obligatoire)* – Gestion des proies

1. Écrivez une fonction de mise à jour des proies. Cette fonction devra :
   — faire bouger les proies en appelant la fonction `bouger_animaux` ;
   — parcourir la liste de proies :
      — baisser leur énergie de 1,
      — manger de l'herbe s'il y en a, gagner `gain_energie_proie` points d'énergie, mettre le temps de repousse de l'herbe de la case où est l'animal à `temps_repousse_herbe` (qui est négatif).
      — supprimer les proies dont l'énergie est inférieure à 0 ;
   — faire appel à la fonction de reproduction.
   La fonction aura le prototype suivant :
   ```
   void rafraichir_proies(Animal **liste_proie, int monde[SIZE_X][SIZE_Y]);
   ```
2. Dans la fonction main du fichier `main_ecosys.c`, vous créerez 20 proies puis vous écrirez une boucle qui s'arrête lorsqu'il n'y a plus de proies ou qu'un nombre maximal d'itération est atteint (par exemple 200). Dans cette boucle vous mettrez à jour les proies et afficherez l'écosystème résultant.
   Pour que vous ayez le temps de voir l'état de votre écosystème, vous pourrez ajouter des pauses en utilisant la fonction `usleep`.
   ```
   man 3 usleep
   ```
   pour avoir une description de son fonctionnement et de la façon de l'utiliser.

Exercice 8 *(entraînement)* – Gestion des prédateurs

1. Pour gérer les prédateurs, nous aurons besoin d'une fonction qui va vérifier s'il y a une proie sur une case donnée. Ecrivez cette fonction qui aura le prototype suivant :
   ```
   Animal *animal_en_XY(Animal *l, int x, int y);
   ```
   `l` est la liste chaînée des proies et la valeur renvoyée est un pointeur sur une proie dont les coordonnées sont `x` et `y` (la première rencontrée) ou `NULL` sinon.
2. Les prédateurs sont gérés comme les proies. Comme ils utilisent la même structure, les fonctions `creer_animal`, `ajouter_en_tete_animal`, etc. peuvent être réutilisées telles quelles. Écrivez la fonction de mise à jour de prédateurs qui respectera le prototype suivant :
   ```
   void rafraichir_predateurs(Animal **liste_predateur, Animal **liste_proie);
   ```
   Cette fonction est directement inspirée de la fonction de rafraichissement des proies. Vous pourrez copier-coller celle-ci et faire des modifications. Vous devrez notamment baisser l'énergie d'un prédateur et faire en sorte que, s'il y a une proie située sur la même case qu'un prédateur, elle soit "mangée" avec une probabilité `p_manger`.
3. Modifier votre fonction main (`main_ecosys.c`) pour voir évoluer également les prédateurs.
   Vous pouvez à présent observer l'évolution de votre petit écosystème et notamment tester différentes valeurs des constantes pour étudier l'impact que cela peut avoir sur votre écosystème.

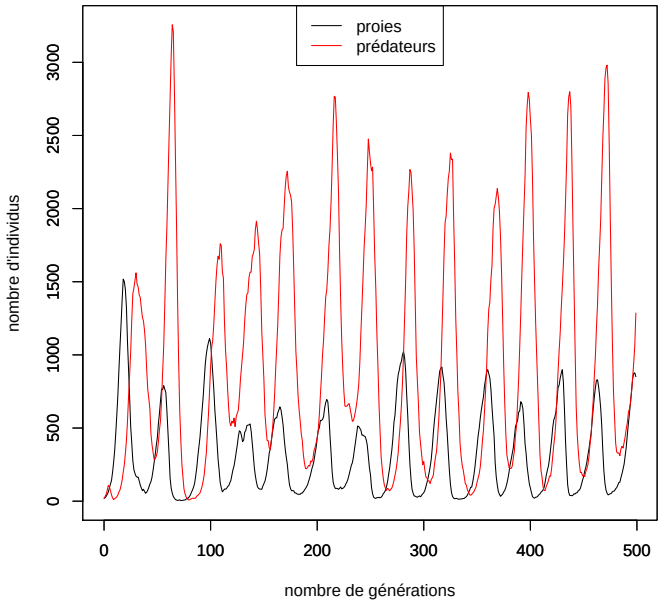
Exercice 9 *(obligatoire)* – Graphiques de l'évolution des populations

Comme vous l'avez peut-être remarqué, et si vos paramètres le permettent, le nombre d'individus oscille au cours des générations. Nous voulons tracer le graphique du nombre d'individus (proies, et prédateurs si vous les avez simulés) en fonction du nombre d'itérations. Pour cela, il faut écrire dans un fichier à chaque itération une ligne contenant l'indice de l'itération, le nombre de proies, le nombre de prédateurs, séparés par des espaces.

1. Dans le fichier `main_ecosys.c` modifiez votre main pour réaliser cela. Vous pourrez ensuite tracer vos courbes dans un tableur ou avec gnuplot en tapant `gnuplot` puis en tapant

   ```
   plot "Evol_Pop.txt"~using 1:2 with lines title "proies"
   replot "Evol_Pop.txt"~using 1:3 with lines title "predateurs"
   ```

   si vous avez écrit les comptages dans un fichier nommé `Evol_Pop.txt`.
   Vous devriez obtenir une courbe ressemblant à cela :



2. Faites varier les paramètres du modèle et observez les variations de population.

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 4
Arbres de mots

Version du 16 septembre 2020

Objectif(s)

- ✯ Arbres binaires de recherche,
- ✯ Arbres lexicographiques,

Exercice(s)

TD : arbres binaires & arbres lexicographiques

Nous allons définir des structures de données et écrire une bibliothèque de fonctions permettant de vérifier, rapidement, si un mot existe ou non dans un ensemble (potentiellement très grand) de mots. Plusieurs implantations possibles seront proposées et comparées.

Le fichier `french_za` vous est fourni. Il contient plus de 130.000 mots de la langue française. Les caractères spécifiques à la langue française (accents et 'ç') ont été enlevés pour simplifier le dictionnaire. Chaque ligne contient un seul mot.

Exercice 1 *(obligatoire)* – Arbre Binaire de Recherche

Une solution très commune pour représenter un dictionnaire consiste à utiliser un arbre binaire de recherche (ABR). À chaque noeud correspond un mot. L'arbre est structuré suivant l'ordre alphabétique des mots qu'il contient :

- Tous les mots qui précèdent le mot porté par un noeud sont dans le sous arbre gauche ;
- Tous les mots qui suivent le mot porté par un noeud sont dans le sous arbre droit ;
- Ces règles sont valables pour tous les noeuds de l'arbre.

Pour représenter les noeuds de cet ABR, nous avons défini la structure de données suivante :

```
typedef struct Nd_mot_ {
  char *mot;
  struct Nd_mot_ *g;
  struct Nd_mot_ *d;
} Nd_mot;
```

Dans le pire des cas la complexité d'un ABR peut équivaloir à celle d'une liste. Pour éviter cet écueil nous allons nous assurer que l'ABR est équilibré par construction. Nous allons pour cela partir d'une liste de mots ordonnée pour construire cet arbre.

L'algorithme est simple, nous allons choisir comme racine de l'ABR le noeud se trouvant au milieu de la liste, utiliser la première moitié de liste pour construire le sous arbre gauche et la seconde pour le sous arbre droit. Cet algorithme est naturellement récursif car il suffira de l'appliquer à nouveau pour construire chacun des sous arbres.

Les similitudes avec l'algorithme du tri rapide sont nombreuses, et comme pour celui-ci la fonction clef sera celle réalisant la partition de la liste en 2 listes. La liste étant déjà ordonnée il aisé de la partitionner équitablement.

La structure de données utilisée pour la liste sera la suivante :

```
typedef struct Lm_mot_ {
  char *mot;
  struct Lm_mot_ *suiv;
} Lm_mot;
```

La fonction suivante vous est fournie :

```
/* taille de la liste donnee en argument */
int taille_Lmot(Lm_mot *lm);
```

1. Écrivez une fonction permettant de couper une liste en 2, son prototype est le suivant :

   ```
   Lm_mot *part_Lmot(Lm_mot **pl)
   ```

   Cette fonction doit retourner un pointeur sur l'élément pivot qui servira de racine à l'ABR et modifiera la liste transmise en argument pour la scinder en son milieu. Il est nécessaire pour cette fonction d'utiliser un double pointeur pour supporter les cas ou la liste comporte moins de 3 éléments. Dans ce cas, le pivot sera le premier élément et vous indiquerez que la liste transmise en argument est maintenant vide.

   Le pointeur retourné correspondra à l'élément pivot et à partir de son champ `suiv` vous pourrez accéder aux éléments de valeurs supérieures. La liste transmise en argument est modifiée par la fonction et ne contiendra plus que les valeurs précédant le pivot.

2. Écrivez maintenant la fonction qui, en partant de la liste ordonnée de mots la transforme en ABR. Au fur et à mesure de son exécution les éléments de la liste sont détruits, les noeuds crées et les chaînes de caractères correspondant aux mots sont réutilisées. Cette fonction est intrinsèquement récursive, elle procède par dichotomie de la liste, créant à chaque exécution un noeud correspondant à l'élément pivot dans la liste qui lui est détruit. Lors du premier appel le noeud créé correspondra à la racine de l'ABR. Après exécution de cette fonction l'ABR sera construit et la liste détruite, il ne sera donc pas nécessaire de libérer ultérieurement la mémoire correspondant à cette liste. Son prototype est le suivant :

   ```
   Nd_mot *Lm2abr(Lm_mot *l);
   ```

3. Écrivez une fonction permettant de rechercher un mot dans l'ABR. Son prototype est le suivant :

   ```
   Nd_mot *chercher_Nd_mot(Nd_mot *abr, const char *mot);
   ```

4. Écrivez une fonction qui libère la mémoire correspondant à l'arbre binaire de recherche. Son prototype est le suivant :

   ```
   void detruire_abr_mot(Nd_mot *abr);
   ```

5. Quelle est, approximativement, la taille en mémoire de l'ABR contenant tous ces mots exprimées en fonction du nombre de mots et de leur longueur moyenne ? Combien de comparaisons faut-il faire lors d'une recherche d'un mot ?

Exercice 2 *(obligatoire)* – Arbre lexicographique

On peut utiliser une structure différente pour stocker un dictionnaire. Plutôt que de stocker chaque mot séparément, on peut construire un arbre dans lequel chaque noeud contient une lettre. Les noeuds fils contiennent la lettre suivante dans le mot et les noeuds frères les autres lettres possibles à cet emplacement. La figure 1 contient un exemple d'arbre lexicographique. Le '.' initial permet de relier toutes les initiales des mots dans un même arbre plutôt que de construire un arbre séparé pour chacune d'elles.

Chaque lettre a jusqu'à 26 successeurs. Nous pouvons le représenter sous la forme d'un arbre binaire. Dans ce cas, le fils gauche contient une des lettres suivantes et le fils droit contient le frère suivant (c'est la représentation d'un arbre général avec un arbre binaire). Avec cette représentation, le noeud '.' n'est plus nécessaire. La figure 2 montre une représentation schématique d'un tel arbre binaire pour stocker les mêmes mots que dans l'arbre de la figure 1.

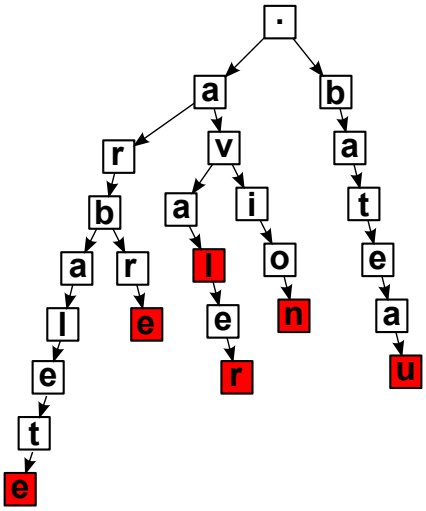


FIGURE 1 – Exemple d'arbre lexicographique contenant les mots "arbalete", "arbre", "aval", "avaler", "avion", "bateau". Les noeuds contenant la lettre finale d'un mot sont colorés.

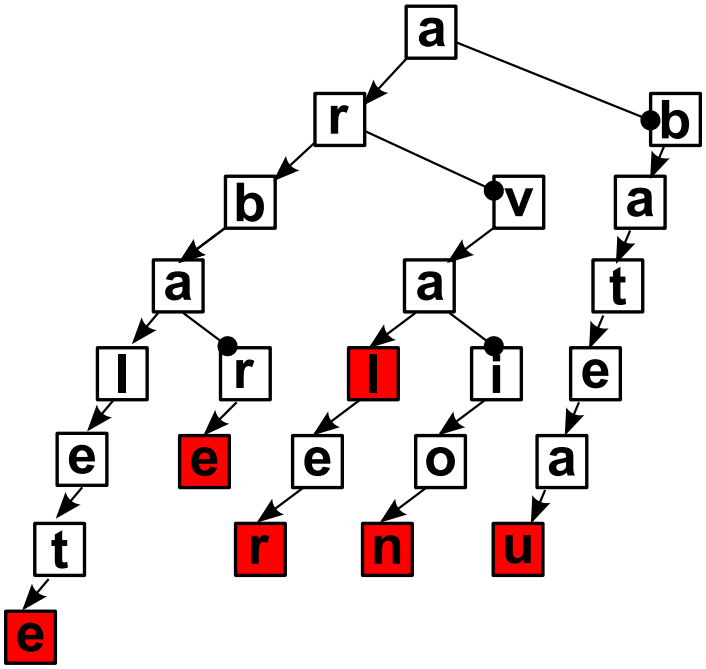


FIGURE 2 – Arbre lexicographique représenté avec un arbre binaire. Cet arbre contient les mêmes mots que sur la figure 1. Les arêtes se terminant par une flèche indiquent une lettre suivante (noeud fils) et les liens se terminant par un rond noir indiquent une lettre alternative (noeud frère). Les noeuds colorés indiquent que le noeud correspondant est la lettre finale d'un mot.

Pour stocker un tel arbre lexicographique, nous avons défini la structure de données suivante :

```
typedef struct noeud *PNoeud;
typedef struct noeud {
  char lettre;
  FDM fin_de_mot;
  PNoeud fils;
  PNoeud frere_suivant;
} Noeud;
```

Avec `FDM` une énumération définie ainsi :

```
// fin = il existe un mot finissant par cette lettre
// non_fin = il n'existe pas de mot finnissant par cette lettre
typedef enum _FDM {fin,non_fin} FDM;
```

1. Écrivez une fonction permettant d'allouer la mémoire associée à un noeud de l'arbre et d'initialiser ses différents champs. Le prototype est le suivant :

   ```
   PNoeud creer_noeud(char lettre);
   ```

2. Écrivez la fonction de recherche d'un mot dans le dictionnaire. Le prototype est le suivant :

   ```
   int rechercher_mot(PNoeud dico, char *mot);
   ```

   Vous pourrez écrire une fonction dédiée à la recherche d'une lettre dans la liste des frères. Cette fonction pourra renvoyer le pointeur vers le noeud trouvé (ou `NULL` sinon).

3. Nous allons maintenant écrire la fonction permettant d'ajouter un mot. Comme les fonctions précédentes, cet ajout se fera par récursion, lettre par lettre. Les lettres absentes de l'arbre seront ajoutées lorsque nécessaire (dans l'ordre alphabétique). Il ne faudra pas oublier de positionner le champ de fin de mot à la fin de l'ajout. Le prototype est le suivant :

   ```
   PNoeud ajouter_mot(PNoeud racine, char *mot);
   ```

   Plutôt que d'utiliser la fonction de recherche dans la liste des frères, vous pourrez écrire une autre fonction qui fera à la fois la recherche d'une lettre dans la liste des frères et l'ajout de la lettre en place si nécessaire.

4. Quelle est, très approximativement, la taille en mémoire de l'arbre contenant tous les mots d'un dictionnaire ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ?

TME : arbres & mots

Exercice 3 *(obligatoire)* – Arbre binaire de recherche, suite

1. Écrivez dans un fichier `main_abr.c` un programme permettant de tester votre dictionnaire *abr*. Dans ce programme, vous testerez la validité de votre bibliothèque en recherchant des mots présents dans le dictionnaire ou non.

   Pour tester la performance (en terme de temps de recherche) de ce dictionnaire vous allez répéter ces opérations un nombre important de fois (plusieurs milliers ou millions de fois) et mesurer le temps à l'aide de la commande `time` dans le terminal.

   La fonction main recevra en argument (ligne de commande) le mot à chercher et nombre de répétitions de la recherche à effectuer. Son prototype est donc le suivant :

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

   Pour transformer l'argument correspondant au nombre de répétitions d'une chaîne de caractères à un entier, vous pourrez utiliser la fonction `atoi`.

   Vous pourrez lire dans le fichier contenant le dictionnaire la liste de mots à partir de laquelle l'arbre sera construit avec la fonction suivante (fournie) :

   ```
   /* Initialisation d'une liste chainee de mots depuis
      un fichier contenant un ensemble de mots */
   Lm_mot *lire_dico_Lmot(const char *nom_fichier);
   ```

Exercice 4 *(obligatoire)* – Arbre lexicographique, suite

1. Écrivez une fonction permettant d'afficher tous les mots stockés dans un arbre lexicographique (fichier `arbre_lexicograp`
   Le prototype est le suivant :
   ```
   void afficher_dico(PNoeud racine);
   ```
   L'affichage d'un mot se fera de la façon suivante : pendant le parcours de l'arbre, une chaîne de caractères est remplie pour garder en mémoire les lettres précédentes. Le mot est affiché lorsque le champ `fin_de_mot` est positionné sur `fin`. Vous pourrez donc utiliser une fonction intermédiaire de prototype :
   ```
   void afficher_mots(PNoeud n, char mot_en_cours[], int index);
   ```
   `mot_en_cours` est la chaîne dans laquelle les lettres sont stockées, `index` indique la position à laquelle écrire la prochaine lettre dans `mot_en_cours`.
2. Écrivez une fonction permettant de libérer la mémoire associée à un dictionnaire. Le prototype est le suivant :
   ```
   void detruire_dico(PNoeud dico);
   ```
3. Écrivez la fonction de lecture d'un dictionnaire depuis un fichier. Le prototype de la fonction est le suivant :
   ```
   PNoeud lire_dico(const char *nom_fichier);
   ```
4. Écrivez dans un autre fichier `main_arbre.c`, un programme permettant de tester votre dictionnaire utilisant des arbres lexicographiques. Comme précédemment, vous indiquerez par les arguments transmis au programme le mot que vous souhaitez chercher. Comme précédemment, pour tester la performance (en terme de temps de recherche) de ce dictionnaire vous allez répéter ces opérations un nombre important de fois (plusieurs milliers ou millions de fois) et mesurer le temps à l'aide de la commande `time`.
   La fonction main recevra en argument (ligne de commande) le nombre de répétitions de la recherche à effectuer. Son prototype est donc le suivant :
   ```
   int main(int argc, char **argv)
   ```

Exercice 5 *(obligatoire)* – Comparaison

1. Les différents programmes de test que vous réalisés vous permettent de comparer en pratique l'efficacité des différents dictionnaires que vous avez implémentés. Quelles sont vos conclusions ?

Exercice 6 *(entraînement)* – Arbre lexicographique bis

Dans la représentation utilisée précédemment, la recherche peut être ralentie par le fait qu'il faut parcourir la liste des noeuds frères avant de trouver le bon. L'utilisation d'une liste chaînée pour stocker les frères a l'avantage de réduire la taille de la structure (seuls les noeuds utiles sont créés), mais cela a un coût en terme d'efficacité : il faut allouer les noeuds un par un et parcourir la liste pour trouver le bon. Nous allons définir une autre structure d'arbre lexicographique. Cette fois, tous les noeuds auront un tableau de 26 fils, un par lettre de l'alphabet (voir figure 3).

La structure de données sera la suivante :

```
typedef struct noeudTab *PNoeudTab;
typedef struct noeudTab {
  char lettre;
  FDM fin_de_mot;
  PNoeudTab fils[26];
} NoeudTab;
```

1. Écrivez, dans le fichier `arbre_lexicographique_tab.c`, les fonctions adaptées à cette nouvelle structure de données. Les prototypes sont les suivants :

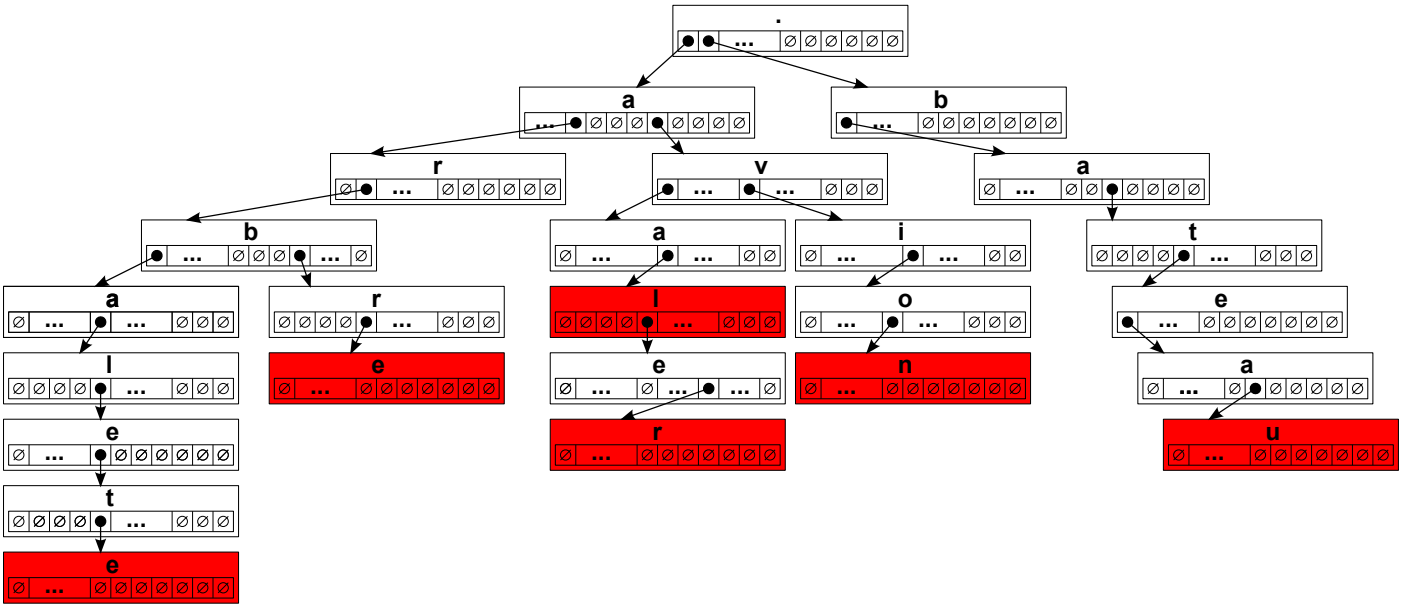


FIGURE 3 – Arbre lexicographique représenté avec des noeuds contenant un tableau de pointeurs sur les noeuds fils. Le symbole ∅ indique que la case correspondante du tableau contient la valeur `NULL`.

```
PNoeudTab creer_noeud(char lettre);
PNoeudTab ajouter_mot(PNoeudTab racine, char *mot);
void afficher_mots(PNoeudTab n, char mot_en_cours[LONGUEUR_MAX_MOT], int index);
void afficher_dico(PNoeudTab racine);
void detruire_dico(PNoeudTab dico);
int rechercher_mot(PNoeudTab dico, char *mot);
```

Ces fonctions doivent se comporter de façon similaire aux fonctions que vous avez écrites avec l'autre implantation de l'arbre lexicographique.

2. En vous inspirant fortement des ce que vous avez fait avant, écrivez la fonction de lecture d'un dictionnaire depuis un fichier pour cette structure de données ainsi qu'un main (fichier `main_arbre_tab.c`).
3. Quelle est, toujours approximativement, la taille mémoire occupée par l'arbre contenant tous les mots d'un dictionnaire ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ?
4. Comparez les résultats obtenus avec cette structure aux structures définies précédemment. Quelles sont vos conclusions ?

TD/TME Semaine 5
Bibliothèque générique de liste

Version du 16 septembre 2020

Objectif(s)

- ✯ Comprendre les pointeurs génériques et les pointeurs de fonction
- ✯ Savoir écrire une bibliothèque générique à partir de code existant
- ✯ Savoir utiliser une bibliothèque générique

Exercice(s)

TD : Conception d'une bibliothèque de liste chaînée générique

Exercice 1 *(base)* – Structures de données

Les principales fonctions de manipulation de liste vues jusqu'à présent sont les suivantes : `inserer_debut`, `inserer_fin`, `inserer_place`, `chercher`, `detruire_liste`, `afficher_liste`, `ecrire_liste` et `lire_liste`.

1. Rappelez ce que font ces différentes fonctions et identifiez celles qui ont besoin de connaitre la donnée portée par un élément de la liste.
2. Déduisez-en les fonctions de manipulation de donnée nécessaires auxquelles les fonctions évoquées ci-dessus pourront faire appel. Proposez-en des prototypes génériques, permettant de s'adapter à n'importe quel type de donnée.
3. Définissez le type décrivant un élément de la liste.

   Les fonctions de manipulation des données peuvent être définies de façon unique dans un programme. Cette solution a l'inconvénient d'empêcher d'utiliser, dans un même programme, notre structure de liste pour des données de type variées. Il ne serait ainsi pas possible de manipuler des listes d'entiers et des listes de chaînes de caractères en même temps, ce qui limiterait notre bibliothèque de liste.
4. Comment pourrait-on procéder pour pouvoir utiliser, dans un même programme, des listes chaînées de types divers ?
5. Proposez la structure de données `Liste` correspondante (`PListe` étant définie comme un pointeur sur `Liste`).

Exercice 2 *(base)* – Implémentation des fonctions de manipulation de la liste

Vous allez maintenant implémenter les fonctions de manipulation de liste. Vous prendrez soin de n'utiliser que les fonctions de manipulation des données évoquées précédemment.

1. Ecrivez la fonction d'insertion en début de liste. Prototype :

   ```
   void inserer_debut(PListe pliste, void *data);
   ```

   La fonction devra dupliquer la donnée passée en argument.

2. Ecrivez la fonction d'insertion en fin de liste. Prototype :
```
void inserer_fin(PListe pliste, void *data);
```
La fonction devra dupliquer la donnée passée en argument.
3. Ecrivez la fonction d'insertion en place. Prototype :
```
void inserer_place(PListe pliste, void *data);
```
La fonction devra dupliquer la donnée passée en argument. Elle ne fera rien si la donnée est déjà dans la liste.
4. Ecrivez la fonction de recherche dans la liste. Prototype :
```
PElement chercher_liste(PListe pliste, void *data);
```
5. Ecrivez la fonction de destruction de la liste. Prototype :
```
void detruire_liste(PListe pliste);
```
6. Ecrivez la fonction d'affichage de la liste. Prototype :
```
void afficher_liste(PListe pliste);
```
La fonction ira à la ligne suivante après chaque donnée.
7. Ecrivez une fonction permettant d'ajouter un nombre quelconque de données en utilisant le principe des fonctions variadiques. Prototype :
```
void ajouter_liste(PListe pliste, int nb_data, ...);
```
`pliste` est la liste à compléter, `nb_data` est le nombre de données à ajouter (c'est le nombre d'arguments après celui-ci).
8. Ecrivez une fonction permettant d'appliquer une même fonction à chaque élément de la liste. Prototype :
```
void map(PListe pliste, void (*fonction)(void *data, void *oa), void *optarg);
```
`pliste` est la liste à considérer, `fonction` est la fonction à appliquer et `optarg` est un argument supplémentaire qui peut être utilisé pour transmettre des informations supplémentaires ou récupérer un résultat de traitement.
Nous allons à présent gérer la lecture et l'écriture dans des fichiers. Pour pouvoir lire une liste, il faudra, à un moment ou à un autre, connaître le type des données qu'elle contient. Nous pouvons soit définir un code associé à chaque donnée, mais cela nécessite de disposer d'une table de tous les codes possibles, ce qui n'est pas facile à maintenir. L'alternative que nous choisirons ici s'appuie sur l'utilisateur et sur le fait qu'il connait à l'avance le type des données qu'il va lire. Lors de l'écriture, il ne sera donc pas nécessaire d'ajouter de code indiquant le type des données, par contre, lors de la lecture, il faudra transmettre une liste qui ne contiendra aucun élément, mais permettra de trouver les fonctions appropriées pour manipuler les données.
9. Ecrivez la fonction d'écriture de la liste. Prototype :
```
int ecrire_liste(PListe pliste, const char *nom_fichier);
```
La fonction renverra 0 s'il y a une erreur ou 1 si tout s'est bien passé.
10. Ecrivez la fonction de lecture de la liste. Prototype :
```
int lire_liste(PListe pliste, const char * nom_fichier);
```
La fonction renverra 0 s'il y a une erreur ou 1 si tout s'est bien passé.

Exercice 3 *(obligatoire)* – Fonctions de manipulation d'entiers

1. Ecrivez les fonctions de manipulations d'entiers. Prototypes :
```
void *dupliquer_int(const void *src);
void copier_int(const void *src, void *dst);
void detruire_int(void *data);
void afficher_int(const void *data);
int comparer_int(const void *a, const void *b);
int ecrire_int(const void *data, FILE *f);
void * lire_int(FILE *);
```

TME : Utilisation de la bibliothèque

Récupérez les fichiers fournis pour cette séance. Ils contiennent les implémentations des fonctions vues en TD.

Exercice 4 *(obligatoire)* – Test de la bibliothèque avec des entiers

1. Ecrivez une fonction main (fichier `ex_liste_entiers.c`) pour tester la bibliothèque de liste avec des données entières. Vous prendrez soin d'utiliser chacune des fonctions de manipulation de la liste et de faire en sorte de vérifier le résultat de l'appel.

Exercice 5 *(obligatoire)* – Listes de chaînes de caractères

1. Ecrivez les fonctions de manipulations de chaines de caractères (fichier `fonctions_string.c`). Prototypes :

```
void *dupliquer_str(const void *src);
void copier_str(const void *src, void *dst);
void detruire_str(void *data);
void afficher_str(const void *data);
int comparer_str(const void *a, const void *b);
int ecrire_str(const void *data, FILE *f);
void * lire_str(FILE *);
```

2. Ecrivez une fonction main pour tester la bibliothèque de liste avec des données de type chaînes de caractères (fichier `ex_liste_string.c`). Vous prendrez soin d'utiliser chacune des fonctions de manipulation de la liste et de faire en sorte de vérifier le résultat de l'appel.

Exercice 6 *(entraînement)* – Manipulation d'un dictionnaire

Nous allons maintenant écrire une fonction permettant de mesurer la longueur des mots contenus dans un dictionnaire. Pour cela, nous allons écrire les fonctions de manipulation d'une donnée composée de deux entiers. Nous nous en servirons pour stocker le résultat de notre comptage qui sera une liste de ces ensembles de 2 entiers : le premier entier sera une longueur et le second le nombre de mots de cette longueur dans le dictionnaire.

1. Ecrivez les fonctions de manipulations de cette donnée avec 2 entiers (fichier `fonctions_2entiers.h`). Structure et prototypes :

```
typedef struct double_int {
  int a;
  int b;
} Double_int;


void *dupliquer_2int(const void *src);
void copier_2int(const void *src, void *dst);
void detruire_2int(void *data);
void afficher_2int(const void *data);
int comparer_2int(const void *a, const void *b);
int ecrire_2int(const void *data, FILE *f);
void * lire_2int(FILE *);
```

La donnée sera de type `Double_int`.

L'affichage se fera sous la forme suivante :

```
a=12 b=42
```

La comparaison ne s'appuiera que sur le champ `a`.

2. Pour compter le nombre de mots, nous allons nous appuyer sur la fonction `map`. Nous devons donc écrire la fonction de traitement d'un mot de la liste chaînée. Cette fonction, qui gèrera le comptage, va recevoir en argument un `char *` sous la forme d'un pointeur générique `void *`. L'argument optionnel va nous servir à stocker le résultat sous la forme d'une liste de double entiers (la longueur et le nombre de mots de cette longueur). Cette liste sera créée avant de faire appel à la fonction `map`. La fonction de traitement d'un mot mesurera la longueur du mot transmis et cherchera si cette longueur fait partie de la liste de résultats actuels, si oui, le nombre d'éléments de cette taille sera incrémenté, sinon, on ajoutera cette longueur avec la valeur 1 dans la liste (elle sera ajoutée en place).

   Ecrivez la fonction de comptage et la fonction main permettant de charger un dictionnaire (fichier `french_za` fourni), de calculer le nombre de mots avec leur longueur, d'afficher le résultat et de libérer la mémoire (fichier `compte_mots.c`).

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 6
Bibliothèque générique d'arbres

Version du 16 septembre 2020

Objectif(s)

- ✯ Approfondir les arbres (binaires, génériques)
- ✯ Approfondir les bibliothèques génériques (écriture, utilisation)

Exercice(s)

TD : Conception d'une bibliothèque générique d'arbres

Dans ce TD, vous allez concevoir une bibliothèque d'arbres qui permet de gérer n'importe quel type de données. Comme pour les listes, vous aurez donc à définir les structures de données, les fonctions nécessaires pour manipuler les données (qui devront être définies pour chaque nouveau type à gérer), et des fonctions génériques qui réaliseront leur traitement en s'appuyant, si besoin, sur ces fonctions spécifiques aux données.

Exercice 1 *(base)* – Arbres binaires

1. Quelles fonctions de manipulation des données stockées dans un noeud seront nécessaires ?
2. En vous inspirant de ce qui a été fait pour les listes génériques, définissez la ou les structures de données des arbres binaires génériques. Vous ajouterez un champ `copie` de type `char` à la structure d'arbre pour indiquer si les données doivent être dupliquées lors de la création d'un noeud.
3. Écrivez la fonction de création d'un arbre binaire vide.
4. Écrivez la fonction de création d'un noeud. La donnée `data` ne sera dupliquée que si le champ `copie` de l'arbre est différent de 0. Prototype :

   ```
   PNoeudBinaire creer_noeud_binaire(PArbreBinaire pab, void *data);
   ```

   Le noeud ne sera pas ajouté à l'arbre. Le pointeur sur l'arbre permet juste de récupérer la fonction de duplication de la donnée, si besoin.
5. Écrivez la fonction de destruction d'un arbre. Vous prendrez en compte le champ `copie` pour libérer ou non la mémoire sur laquelle pointe `data`. Vous ferez une fonction de destruction de l'arbre qui fera appel à une fonction récursive que vous écrirez également.
   Prototype :

   ```
   void detruire_ab(PArbreBinaire pab);
   ```

6. Écrivez la fonction d'affichage d'un arbre avec un parcours préfixe. Comme pour la question précédente, vous ferez une fonction d'affichage qui fera appel à une fonction récursive que vous écrirez également. Prototype :

```
void afficher_ab_prefixe(PArbreBinaire pab);
```

   Vous pourrez éventuellement ajouter des "(" et des ")" pour indiquer la structure de l'arbre. Exemple :

```
( ( 1( ( ( 17) 22( 44) ) 52( 64) ) ) 72( 85( ( 86) 98) ) )
```

7. Que faudrait-il changer à cette fonction pour obtenir un affichage avec un parcours en infixe ? En postfixe ?

Exercice 2 *(obligatoire)* – Arbres binaires de recherche

Les fonctions écrites jusqu'à présent ne faisaient aucune hypothèse particulière sur la façon d'organiser l'arbre (mis à part le fait que c'était un arbre binaire). Vous allez maintenant écrire les fonctions permettant de gérer un arbre binaire de recherche. Pour cela il suffira d'écrire la fonction d'ajout dans l'arbre et la fonction de recherche.

1. Écrivez la fonction d'ajout dans l'arbre. Pour rappel, dans un arbre binaire de recherche, le sous-arbre gauche contient les données inférieures au noeud courant et le sous-arbre droit contient les données supérieures. Lors de l'ajout d'une donnée, il faut donc parcourir l'arbre en continuant dans le sous-arbre gauche ou droit jusqu'à atteindre la fin de l'arbre. La donnée est alors ajoutée dans une feuille créée à cette occasion. La fonction ne fait rien si la donnée est déjà dans l'arbre. Prototype :

```
void ajouter_abr(PArbreBinaire pab,void *data);
```

   Comme précédemment, cette fonction fera appel à une fonction récursive que vous écrirez également.

2. Écrivez la fonction de recherche dans l'arbre. Prototype :

```
PNoeudBinaire chercher_abr(PArbreBinaire pab, void *data);
```

3. Écrivez une fonction de lecture de l'arbre depuis un fichier. Chaque élément lu sera inséré dans l'arbre avec la fonction d'ajout écrite ci-dessus. Prototype :

```
void lire_abr(PArbreBinaire pab, const char *nom_fichier);
```

   L'arbre fourni en argument est un arbre vide (dont la racine vaut NULL). Il est transmis pour que la fonction de lecture de l'arbre puisse utiliser la fonction de lecture de donnée appropriée.

   La fonction de lecture alloue la mémoire des données lors de la lecture. Afin d'éviter de dupliquer ces données, pendant l'ajout des éléments lus, vous prendrez soin de mettre le champ `copie` à 0. Vous mettrez ensuite ce champ à 1 pour que ces données soient bien libérées lorsque la mémoire associée à cet arbre sera libérée.

4. Écrivez une fonction d'écriture de l'arbre dans un fichier. Choisissez la méthode de parcours de l'arbre qui permettra à la fonction de lecture ci-dessus de reconstruire l'arbre à l'identique après lecture. Ce parcours sera dans une fonction récursive que vous écrirez également. Prototype :

```
void ecrire_ab(PArbreBinaire pab, const char *nom_fichier);
```

Exercice 3 *(approfondissement)* – Utilisation de la fonction `map`

Un certain nombre des fonctions écrites jusqu'à présent reprennent un même schéma. Pour éviter d'avoir à répéter ce code, vous allez écrire une fonction `map` pour appliquer une fonction donnée en argument à tous les éléments de l'arbre. Vous l'appliquerez ensuite pour redéfinir quelques-unes des fonctions vues jusqu'à présent.

1. Écrivez la fonction `map` permettant d'appliquer à chaque élément de l'arbre, avec un parcours préfixe, une fonction transmise en argument. Comme précédemment, il faut faire une première fonction qui appelle une fonction récursive qu'il faut aussi écrire. Prototype :

```
void map_ab_prefixe(PArbreBinaire pab, void (*fonction)(void *data, void *oa), void *optarg);
```

   `optarg` est un argument permettant de transmettre des arguments supplémentaires à la fonction qui sera appelée sur chaque élément. Cela peut aussi permettre de récupérer une valeur calculée.

2. La fonction map que l'on a définie ci-dessus nécessite, pour plus de souplesse une fonction prenant en argument un `void *` et un argument supplémentaire `optarg`. La fonction d'affichage ne prend qu'un `void *`. Définissez une fonction ayant le prototype approprié pour redéfinir une fonction d'affichage en s'appuyant sur `map`, fonction que vous écrirez également. Astuce : transmettez un pointeur sur l'arbre via `optarg`.
3. Utilisez la fonction `map` pour réécrire la fonction d'écriture dans un fichier. Vous aurez pour cela besoin de définir une structure dédiée à la transmission des informations appropriées via l'argument `optarg` et vous aurez également à écrire la fonction qui sera transmise en argument à `map`.
4. Pouvez-vous écrire une version `map` de la fonction de destruction de l'arbre ? Si non, que faudrait-il faire ?

TME : Utilisation de la bibliothèque

Récupérez les fichiers fournis pour cette séance. Ils contiennent les implémentations des fonctions vues en TD.

Exercice 4 *(base)* – Tests de la bibliothèque d'arbres binaires

1. Écrivez une fonction `main` pour tester la bibliothèque d'arbres (fichier `ex_ab_entiers.c`). Dans un premier temps, vous la testerez avec des entiers. Vous créerez un arbre binaire de recherche dans lequel vous insèrerez une dizaine d'entiers aléatoires compris entre 0 et 99. Vous afficherez l'arbre puis vous ferez une boucle pour chercher tous les entiers compris entre 0 et 99 afin de vérifier le bon fonctionnement de votre fonction de recherche. Vous écrirez l'arbre dans un fichier, puis vous le lirez dans un nouvel arbre que vous afficherez. Vous finirez votre `main` par la libération de toute la mémoire allouée. Vous prendrez soin de tester votre exécutable avec `valgrind` pour vérifier qu'il n'y a pas de fuite mémoire.
2. Écrivez une fonction main pour tester la bibliothèque d'arbres avec des mots (fichier `ex_ab_mots.c`). Vous créerez un arbre binaire de recherche dans lequel vous insèrerez les mots lus depuis le dictionnaire contenu dans le fichier "french_za_reordered". Vous rechercherez quelques mots qui sont ou ne sont pas dans le dictionnaire pour vérifier le bon fonctionnement de la bibliothèque. Vous pourrez également l'écrire dans un autre fichier et vérifier que le contenu est le même. Choisissez la fonction d'écriture permettant de récupérer le fichier trié par ordre alphabétique. Pour la comparaison, vous pourrez utiliser la commande shell `diff` qui permet de comparer deux fichiers et la commande shell `sort` qui permet de trier un fichier (pour générer une version triée de "french_za_reordered") . Vous prendrez soin de tester votre exécutable avec `valgrind` pour vérifier qu'il n'y a pas de fuite mémoire.

Exercice 5 *(obligatoire)* – Arbres d'expressions

Vous allez maintenant utiliser la bibliothèque générique d'arbres pour implémenter des arbres d'expression. Un arbre d'expression permet de représenter une expression arithmétique. Les noeuds internes de l'arbre correspondent à des opérateurs, qui ne seront ici que des opérateurs binaires (ce qui permettra de représenter ces expressions avec des arbres binaires). Les feuilles de l'arbres seront soit des constantes, soit des variables. L'évaluation d'un noeud de l'arbre est sa valeur, s'il s'agit d'une constante ou d'une variable, ou le résultat de l'opération portée par le noeud et appliquée aux opérandes correspondant à leurs sous-arbres gauche et droit. L'évaluation de l'arbre est le résultat de l'évaluation de sa racine. Remarque : l'évaluation de l'arbre nécessite de demander la valeur des variables. La figure 1 présente un exemple d'arbre d'expression. Il représente l'expression arithmétique : $(3 * X) + (Y/6)$.

Un fichier contenant une fonction `main` vous est fourni (fichier `main_expr.c`). Vous pourrez l'utiliser en le modifiant à votre guise pour tester vos fonctions au fur et à mesure.

Les noeuds pourront être de 3 types : constante, variable ou opérateur. Un type enuméré permettra d'identifier le type d'un noeud :

```
typedef enum _type_noeud {VAR, CST, OP} TypeNoeud;
```

La donnée portée par un noeud associera à ce type, les champs nécessaires pour gérer les 3 cas de figure :

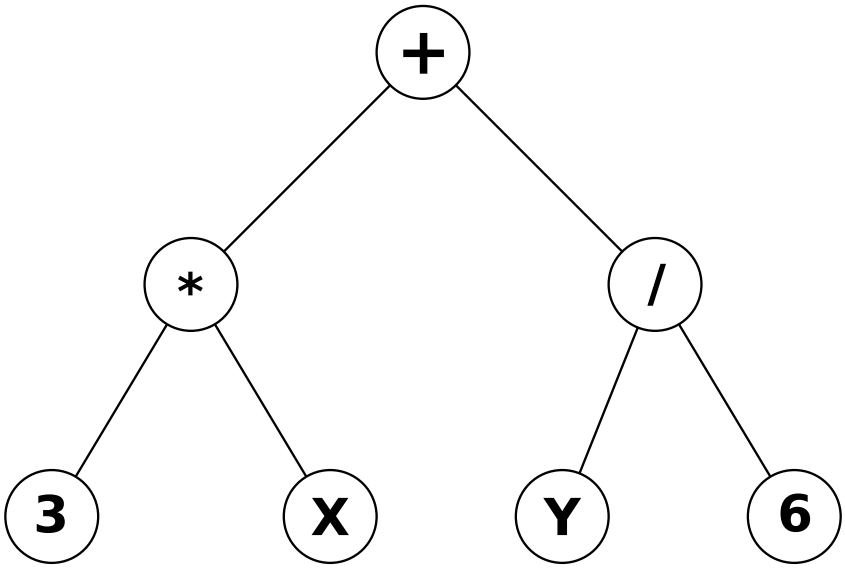


FIGURE 1 – Exemple d'arbre d'expression.

```
typedef struct _dataExpr
{
  TypeNoeud type_noeud;
  unsigned int indice_var;
  char operateur;
  float constante;
} DataExpr;
```

Nous ne considèrerons que les variables représentées par une lettre majuscule. Le champ associé à une variable est l'indice de cette variable : 0 correspond à la variable A, 1 à B, ... Un opérateur est représenté par un caractère. Nous ne considèrerons que les 4 opérateurs suivants : addition (représentée par un '+'), soustraction ('-'), multiplication ('*') et division ('/').

1. En vue de leur utilisation dans la bibliothèque d'arbres génériques, écrivez les fonctions permettant de dupliquer, copier et détruire de telles données (fichier `arbre_expr.c`). Prototypes :

   ```
   void *dupliquer_expr(const void *src);
   void copier_expr(const void *src, void *dst);
   void detruire_expr(void *data);
   ```

   Remarque : nous n'utiliserons pas la fonction de comparaison, il suffit donc de définir une fonction renvoyant, par exemple, toujours 0.

2. Écrivez la fonction d'écriture d'une donnée. Cette fonction écrira, selon le type de la donnée, soit la variable ('A' si l'indice est 0, 'B' si c'est 1, ...), soit la constante, soit le caractère représentant l'opération. Un espace sera ajouté avant et après cet affichage. La fonction d'affichage utilisera le même format. Vous pourrez donc vous contenter de la définir à partir de la fonction d'écriture à qui vous transmettrez `stdout`, qui correspond à la sortie standard. Prototypes :

   ```
   int ecrire_expr(const void *data, FILE *f);

   void afficher_expr(const void *data);
   ```

3. La fonction de lecture d'une donnée nécessite de gérer les 3 différents cas de figure. Plusieurs implémentations sont possibles. L'une d'entre elle consiste à mettre dans un buffer la prochaine donnée à analyser. Lorsque les premiers espaces éventuels ('espace' au sens large de `isspace`, tapez `man 3 isspace` dans un terminal pour en savoir plus) ont été enlevés, il s'agit des caractères avant le prochain 'espace'. Vous pouvez ensuite analyser les premiers caractères pour distinguer les 3 cas de figures et initialiser la donnée lue en fonction de cela. Vous pourrez utiliser les fonctions de test de caractères `isdigit` et `isupper`.
   Prototype :

   ```
   void * lire_expr(FILE *);
   ```

4. La lecture de l'arbre ne peut pas s'appuyer sur la bibliothèque générique, car la fonction vue précédemment est dédiée aux arbres binaires de recherche. La lecture de l'arbre d'expression nécessite la création d'une fonction récursive qui va lire noeud après noeud. La fonction principale de lecture se contentera de créer l'arbre et de faire appel à la fonction récursive. La fonction récursive lira la donnée d'un noeud à l'aide de la fonction vue précédemment et, si le noeud est un opérateur, fera deux appels récurrents à elle-même pour lire les sous-arbres gauche et droit. Comme précédemment, vous penserez à mettre le champ `copie` à 0 pendant la lecture et à 1 ensuite pour que les données soient bien effacées lorsque l'arbre sera détruit.

   Prototype :

```
PNoeudBinaire construire_arbre_expr_rec(FILE *f, PArbreBinaire pab);
PArbreBinaire construire_arbre_expr(FILE *f);
```

5. Écrivez la fonction d'évaluation de l'arbre. Cette fonction reçoit en argument l'arbre d'expression ainsi que le tableau des valeurs des variables contenues dans l'arbre.

   Prototype :

```
float evaluer(PNoeudBinaire pabe, float var_lue[]);
```

