TME Séance 0: Remise en jambe...

Version du 27 août 2024

Objectif(s)

- ✯ Rappels sur la compilation et l'exécution de programmes.
- ✯ Rappels manipulations de tableaux et de structures
- ✯ Introduction à make et aux fichiers Makefile.

Exercice 1 *(base)* – Hello world

1. Écrivez un programme qui affiche "Hello world !" dans le terminal. Vous le mettrez dans un fichier source `hello_world.c`. Vous le compilerez avec `gcc` (votre exécutable s'appellera `hello_world`) avant de l'exécuter dans le terminal pour observer le résultat. Pour rappel voilà les principales options de `gcc` :
   — `-Wall` : afficher tous les "warnings"
   — `-o file` : la sortie de `gcc` est envoyée dans le fichier `file` que l'on peut ensuite exécuter avec `./file`.
2. Modifiez votre programme pour qu'il affiche "Bonjour monde !", compilez-le et exécutez-le de nouveau. Vérifiez que vous voyez bien le message modifié lors de l'exécution de votre programme.
3. Dans le fichier `hello_world2.c`, complétez la fonction `affiche` pour afficher un message texte en ASCII art. Chaque caractère du message sera remplacé par un ensemble de caractères choisi pour sa ressemblance avec le caractère du message. Si par exemple, vous voulez afficher la chaîne de caractères "BONJOUR", vous devrez voir à l'écran :

```
 _     _           _    _          _
|_)   / \   |\|     |   / \   | |   |_)
|_)   \_/   | |   \_/   \_/   \_/   | \
```

   Vous ne pourrez afficher que des caractères majuscules et des espaces. Les caractères représentants les lettres sont dans le tableau `lettre` et ceux du caractère espace sont dans le tableau `espace`.

   Pour afficher le contenu du tableau `message`, vous devrez afficher, ligne par ligne, les caractères de chaque lettre du message. Pour afficher "BONJOUR", par exemple, vous devez afficher la première ligne du tableau représentant 'B', puis la première ligne du tableau représentant 'O', etc. Ensuite vous sautez à la ligne, puis vous affichez la deuxième ligne du tableau représentant 'B', puis la deuxième ligne du tableau représentant 'O', etc.

   Remarque : le tableau du caractère de 'A' est dans la case 0 de `lettre` (il s'agit donc de `lettre[0]`). Pour retrouver l'indice d'une lettre c dans le tableau, vous pouvez donc utiliser c-'A' (il faut que c soit le code d'une lettre majuscule). L'espace devra être traité à part.

   Votre fonction contiendra donc 3 boucles imbriquées : une boucle par ligne (`N_LIGNES` en tout) puis une boucle par caractère du message puis une boucle par caractère de la ligne à afficher (`N_COLONNES` en tout).

   Vous compléterez le main pour afficher "HELLO WORLD".

Exercice 2 *(base)* – Makefile

Lors des questions précédentes, vous avez écrit plusieurs programmes. Chaque fois que vous faites une modification au code source de ces programmes, il faut recréer l'exécutable avec `gcc` avant de le lancer afin de voir le résultat de cette modification. Des outils sont conçus pour vous aider à cela et notamment :

1. créer un exécutable particulier avec une commande courte
2. créer d'un coup tous les exécutables
3. mettre à jour un ou plusieurs exécutables, mais uniquement si c'est nécessaire

Nous allons voir comment utiliser `make` et les fichiers `Makefile` dans ce but. `make` est un outil standard dans le monde du logiciel libre. Il n'est pas lié au langage C (vous pouvez l'utiliser dans d'autres contextes, ce PDF est généré depuis un fichier LaTeX compilé avec `make`, par exemple). Il est très utile pour faciliter le développement en C lorsque l'on n'a pas d'environnement intégré. Nous n'en ferons qu'un usage très basique, il permet bien plus que ce nous utiliserons ici.

`make` est un programme permettant d'automatiser des opérations (par exemple une compilation) en s'appuyant sur un fichier `Makefile` qui spécifie ce qu'il faut faire et dans quelles conditions le faire. Pour utiliser `make`, il suffit donc d'écrire un fichier `Makefile`. Ce fichier est un fichier texte qui contient des déclarations de variables (nous verrons cela plus tard) et des règles.

Les règles sont de la forme suivante :

```
cible: dependance(s)
          action(s)
```

Les cibles sont des fichiers générés par les actions (par exemple votre exécutable).

ATTENTION : les lignes contenant les actions doivent commencer par une tabulation.

Les dépendances sont les fichiers dont dépend votre cible, c'est-à-dire les fichiers dont la modification nécessite de reconstruire la cible.

Lorsque, dans le terminal, vous tapez :

```
terminal$ make cible
```

cela déclenche la règle ayant la cible "cible". S'il n'y a pas de fichier nommé "cible" ou s'il est plus ancien que les dépendances, les actions sont déclenchées.

Exemple : si votre `Makefile` contient la cible :

```
mon_prog: mon_prog.c
          gcc -Wall -o mon_prog mon_prog.c
```

lorsque vous tapez :

```
terminal$ make mon_prog
```

cela déclenchera la compilation de `mon_prog` s'il n'existe pas ou si le fichier `mon_prog.c` a été modifié.

1. Ecrivez un fichier Makefile permettant de compiler votre programme `hello_world`. Effacez l'exécutable (attention à ne pas effacer le fichier C...) et utilisez `make` pour le recréer.
2. Modifiez le message affiché et mettez à jour l'exécutable avec `make`. Vérifiez que l'exécutable n'est pas changé si vous faites appel à `make` et que le fichier C n'a pas été modifié.
3. Complétez votre Makefile pour qu'il puisse créer le deuxième exécutable et vérifiez qu'il permet bien de le créer.
4. Ajoutez, au début du `Makefile`, une cible `all` qui aura pour dépendances tous les exécutables et qui n'aura pas d'action. Cette règle sera lancée chaque fois que l'on tapera `make all` ou `make` (qui par défaut lance la première règle du `Makefile`). Comme il n'y a pas d'action et comme aucun fichier "all" n'est créé, cette règle sera toujours déclenchée. Elle permet de lancer, d'un seul coup, toutes les cibles importantes du `Makefile`.

Créez également une cible `clean` pour effacer les exécutables (attention à ne pas faire d'erreur dans l'écriture de cette ligne...). Cette règle n'aura pas de dépendances. L'action associée sera un `rm` des fichiers générés, donc des exécutables (attention à ne pas supprimer les fichiers source ! !).

Vérifiez que tout fonctionne correctement en utilisant `make` pour effacer les exécutables et les recréer avec un `make` tout court, sans spécifier de cible.

Exercice 3 *(approfondissement)* – Manipulations d'images

Afin de revenir sur l'allocation dynamique et l'utilisation des structures, vous allez écrire des fonctions de manipulations simples d'images. Les fonctions de lecture et d'écriture des images vous sont fournies. Vous verrez les images au travers de la structure de données suivante :

```
typedef struct _image_t
{
  unsigned long w; // largeur en pixels
  unsigned long h; // hauteur en pixels
  unsigned char *buff; // w x h octets correspondant aux pixels
} image_t;
```

L'image est stockée dans le champ `buff` sous la forme d'un tableau d'entiers de type `unsigned char` (donc entre 0 et 255). Chaque entier indique le niveau de gris d'un pixel, 0 étant le noir et 255 le blanc.

1. Ecrivez les fonctions de création et de destruction d'image. La création doit allouer une image avec un `malloc` et initialiser ses champs.Vous laisserez le champ `buff` à NULL et les champs w et h à 0. Il ne faudra pas oublier de libérer le buffer dans la fonction de destruction d'une image. Ce dernier est alloué dynamiquement lors du chargement d'une image qui initialise alors le champ buff avec w*h valeurs.
2. Ecrivez la fonction pour prendre le négatif d'une image. Il suffit de parcourir le champ `buff` et de remplacer la valeur v d'un pixel par 255-v. Vous pouvez modifier directement le buffer de l'image.
3. Compléter la fonction `main` pour tester vos fonctions et ajouter une règle à votre Makefile pour compiler ce nouvel exécutable.

   Vous pourrez utiliser les fonctions de lecture et d'écriture d'images fournies. Elles permettent de lire et d'écrire des images au format PGM, qui est un format simple à lire et écrire. Si vous voulez tester des images autres que celles fournies, vous pouvez les mettre au format PGM avec la commande `convert` dans le terminal :

   ```
   terminal$ convert fichier.jpg fichier.pgm
   ```

4. (**Optionnel**) Ecrivez une fonction de modification de la luminosité d'une image. Vous multiplierez le niveau de gris par le pourcentage transmis en argument à la fonction (qui est un nombre réel, 1.0 signifiant par de changement, 0.5 luminosité moitié, 2.0 luminosité double, etc). Vous prendrez soin de borner la nouvelle valeur à 255 avant de la remettre dans le buffer.
5. (**Optionnel**) Ecrivez une fonction de rotation d'une image. Cette fonction réalise une copie de l'image fournie en argument `src` et applique une des trois rotations possibles à cette image (90 °, 180 ° ou 270 °). La rotation est considérée dans le sens trigonométrique. La fonction devra vérifier la validité de l'argument `angle` fourni. Cette fonction retourne un pointeur sur la nouvelle image. Nous vous conseillons d'utiliser la macro `VAL` pour simplifier l'écriture de votre code. Bien qu'une rotation de 270 ° puisse être obtenue par 3 rotations de 90 °, afin d'optimiser les traitements, chaque rotation possible doit être obtenue directement.

   ```
   #define VAL(img,i,j) (img)->buff[(i)*(img)->w+(j)]
   ```

   Où `i` correspond à l'indice de ligne et `j` à celui de colonne.

TD/TME Séance 1
Pointeurs des chaînes

Version du 27 août 2024

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. 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
```

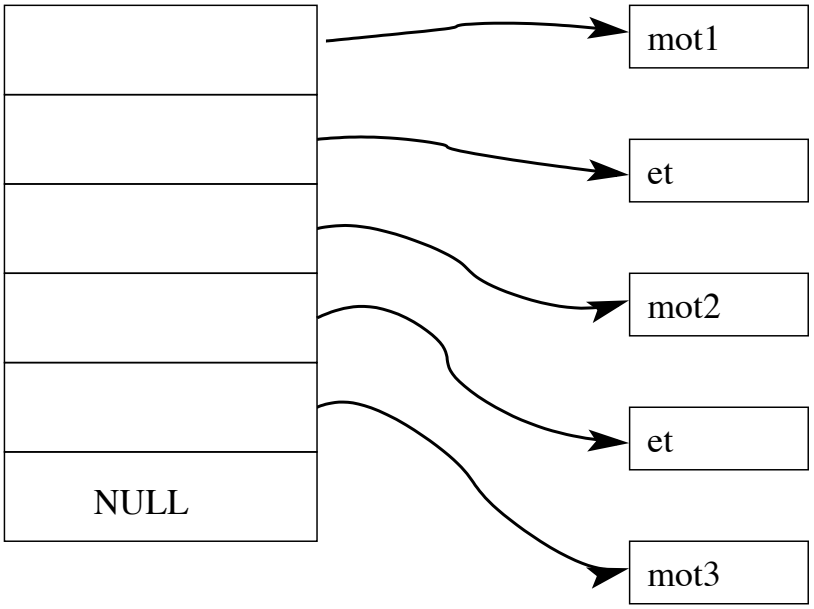



FIGURE 1 – Tableau de pointeurs sur les mots de la chaîne de caractères.

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 par 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.

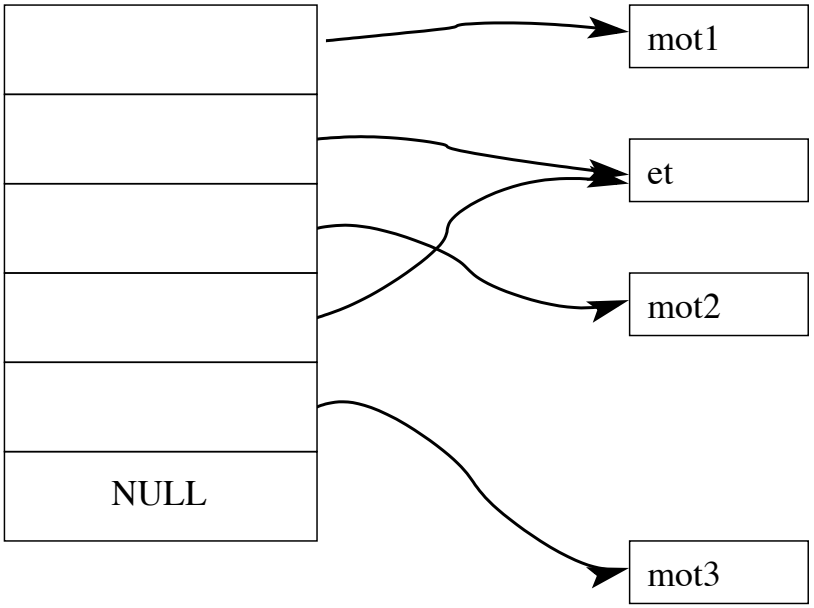


FIGURE 2 – Tableau de chaînes "compacté".

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 ?

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

Version du 27 août 2024

Objectif(s)

- ✯ tableaux 2D statiques en arguments
- ✯ Listes chaînées (rappels)
- ✯ assert
- ✯ Makefile (bis)
- ✯ Bonnes pratiques de programmation : les tests
- ✯ Bonnes pratiques de programmation : ddd/gdb et valgrind

L'objectif de ce sujet est la réalisation d'un projet de programmation d'un écosystème virtuel utilisant des listes chaînées. Nous allons d'abord commencer par quelques exercices de rappel, puis nous aborderons le problème de l'écosystème.

TD : des tableaux aux listes

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)* – Rappels sur les listes chaînées, analyse d'un exemple

Soit la structure de données suivante :

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

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 les proies. 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 ainsi qu'un tableau statique à deux dimensions représentant le monde.

Exercice 3 *(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 4 *(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 5 *(entraînement)* – 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

Exercice 6 *(obligatoire)* – Tests des fonctions et main

Les fonctions évoquées pendant le TD sont fournies, ainsi qu'un main permettant de tester ces fonctions (`main_tests.c`). Un Makefile est également fourni pour créer un exécutable `tests_ecosys`.

1. Il semblerait que les fonctions fournies contiennent des erreurs... Testez le programme, vérifiez si son comportement est correct et corrigez les erreurs éventuelles en vous appuyant sur les outils de débogage qui vous ont été présentés. Vous pouvez également ajouter des `assert` dans les fonctions pour vérifier que les variables ne sortent pas des valeurs ou intervalles attendus. Une bonne pratique consiste à insérer des `assert` chaque fois qu'un bug est découvert pour éviter qu'il ne se reproduise lors des évolutions ultérieures de votre code.
2. Pour faciliter les manipulation des listes de proies et de prédateurs, écrivez une 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 indiquées 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);
```

3. Ecrivez une fonction main dans un fichier `main_tests2.c` qui va créer 20 proies et 20 prédateurs à des positions aléatoires, vérifiez leur nombre en faisant appel aux fonctions de comptage vues précédemment et enfin affichez l'état de votre écosystème.

   Vous complèterez aussi le `Makefile` pour créer le programme `tests_ecosys2`.

4. Ecrivez la fonction `liberer_liste_animaux(Animal *liste)` qui permet de libérer la mémoire allouée pour la liste `liste`.

5. Complétez la fonction `main` de `main_tests2.c` pour libérer toute la mémoire allouée.

6. 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_ecosys2`). 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
   ```

7. 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 Séance 3
Listes chaînées et simulation d'un écosystème - partie 2

Version du 27 août 2024

Objectif(s)

- ✯ lecture et écriture de fichiers
- ✯ débogage et gestion de la mémoire

TD

Exercice 1 *(obligatoire)* – Lecture/écriture de fichiers

Nous allons écrire l'ensemble de l'écosystème dans un fichier pour pouvoir le relire ultérieurement et reprendre l'observation de ses évolutions. Cela peut également aider à déboguer son fonctionnement en comparant l'état de l'écosystème avant et après une mise à jour, par exemple.

Voilà le format du fichier sur un exemple contenant 4 proies et 3 prédateurs :

```
<proies>
x=8 y=43 dir=[0 1] e=10.000000
x=18 y=38 dir=[1 -1] e=10.000000
x=9 y=37 dir=[-1 1] e=10.000000
x=18 y=0 dir=[0 1] e=10.000000
</proies>
<predateurs>
x=14 y=32 dir=[0 -1] e=10.000000
x=13 y=40 dir=[-1 -1] e=10.000000
x=18 y=46 dir=[0 -1] e=10.000000
</predateurs>
```

1. Ecrivez la fonction `ecrire_ecosys` permettant d'écrire ce fichier à partir des listes de proies et de prédateurs. Prototype :

   ```
   void ecrire_ecosys(const char *nom_fichier, Animal *liste_predateur, Animal *liste_proie);
   ```

   La fonction va écrire la ligne `<proies>`, puis toutes les proies, puis `</proies>`, puis `<predateurs>`, puis tous les prédateurs, puis elle termine pas la ligne `</predateurs>` avant de fermer le fichier.

2. Ecrivez la fonction `lire_ecosys` permettant de lire ce fichier et qui retourne les listes de proies et de prédateurs qui ont été lues. Cette fonction prendra en argument le nom du fichier à lire ainsi que les listes de proies et de prédateur par pointeur pour pouvoir les modifier. Ces variables seront supposées pointer sur des listes vides. Prototype :

   ```
   void lire_ecosys(const char *nom_fichier, Animal **liste_predateur, Animal **liste_proie);
   ```


La fonction devra lire ligne par ligne et construire les animaux à partir des éléments lus en les ajoutant à la bonne liste chaînée (proies ou prédateurs).

Nous allons maintenant 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 autant d'énergie qu'il y a d'herbe sur la case ;
— 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 la mangent. 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; //probabilite de changer de direction de deplacement
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 2 *(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. 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);
```

TME

Exercice 3 *(obligatoire)* – Lecture/écriture de fichiers

1. Complétez le main du fichier `main_test2.c` pour écrire l'écosystème créé et le lire dans de nouvelles listes. Vous vérifierez que les écosystèmes lus et écrits sont bien les mêmes. Vous prendrez soin de libérer la mémoire allouée et de vérifier qu'il n'y a pas de fuite mémoire.

Exercice 4 *(obligatoire)* – Déplacement et reproduction

1. Complétez le main du fichier `main_ecosys.c` pour tester la fonction de déplacement. Vous ne créerez 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.
2. Complétez le main du fichier `main_ecosys.c` pour tester la fonction de reproduction. Vous mettrez le taux de reproduction à 1 en vérifierez que le nombre d'animaux est bien multiplié par 2 à chaque mise à jour.

Exercice 5 *(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,
      — 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 6 *(obligatoire)* – 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", permettant ainsi au prédateur de récupérer les points d'énergie de la proie.
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 7 *(entraînement)* – Gestion de l'herbe

Nous pouvons enrichir notre écosystème en ajoutant de l'herbe.Les proies mangent 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.
3. Ajouter la prise en compte de l'herbe dans la fonction de mise à jour des proies. Les proies mangent de l'herbe s'il y en a sur leur case en gagnant autant de points d'énergie qu'il y a d'herbe sur la case. Il faut alors mettre le temps de repousse de l'herbe de la case où est l'animal à `temps_repousse_herbe` (qui est négatif).

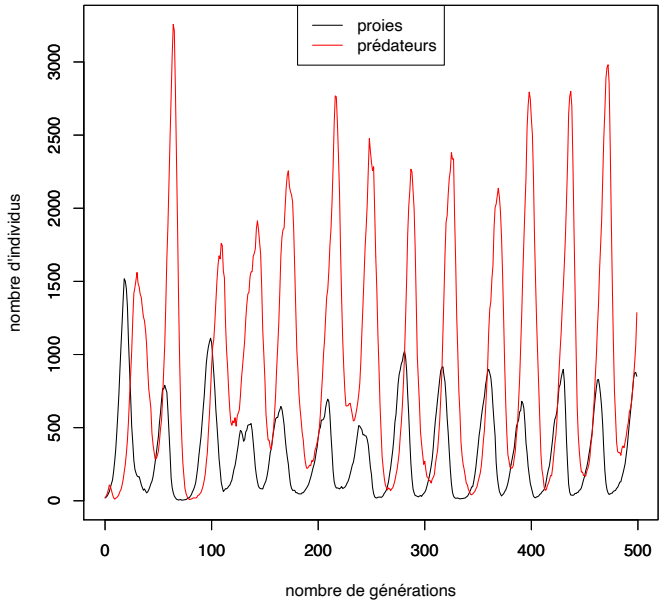
Exercice 8 *(approfondissement)* – Graphiques de l'évolution des populations

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.


proies
prédateurs
nombre d'individus
0
500
1000
1500
2000
2500
3000
0
100
200
300
400
500
nombre de générations

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Séance 4
Arbres de mots

Version du 27 août 2024

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, des algorithmes spécifiques permettent de créer un arbre équilibré (notamment avec des arbres rouge-noir). Nous n'allons pas utiliser ces algorithmes et pour simplifier, nous allons nous assurer que l'ABR est équilibré par construction en le créant à partir d'une liste ordonnée.

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 est 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 où 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 créés 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. Cet élément pivot est lui-même détruit ensuite. 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 (y compris les mots qu'il contient). 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.

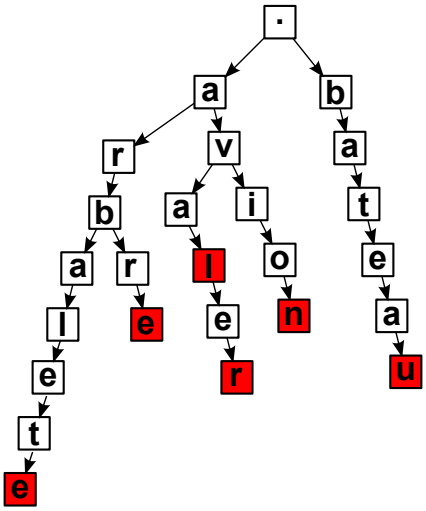


FIGURE 1 – Exemple d'arbre lexicographique contenant les mots "arbalete", "arbre", "aval", "avaler", "avion", "bateau". Les noeuds contenant la lettre finale d'un mot sont colorés.

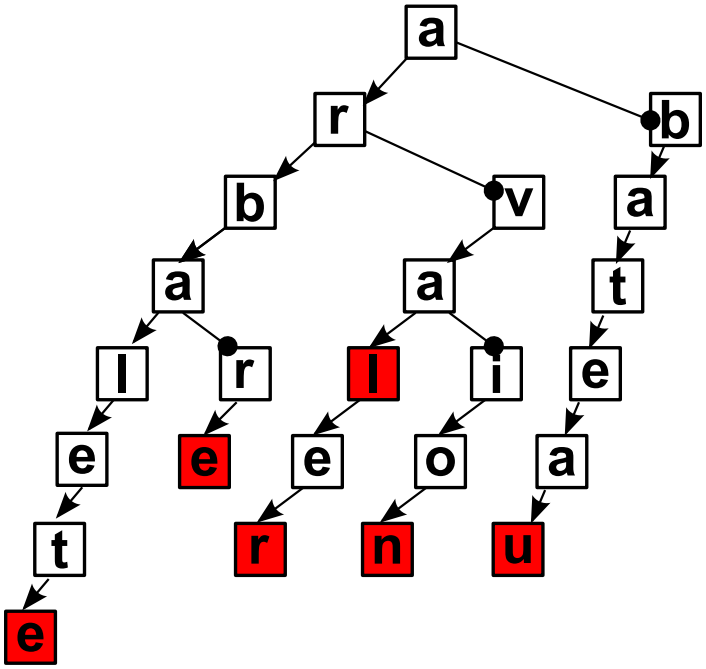


FIGURE 2 – Arbre lexicographique représenté avec un arbre binaire. Cet arbre contient les mêmes mots que sur la figure 1. Les arêtes se terminant par une flèche indiquent une lettre suivante (noeud fils) et les liens se terminant par un rond noir indiquent une lettre alternative (noeud frère). Les noeuds 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 (qui obligerait à faire 2 parcours successifs de l'arbre et serait donc moins efficace), 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_lexicographique.c`). 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 :

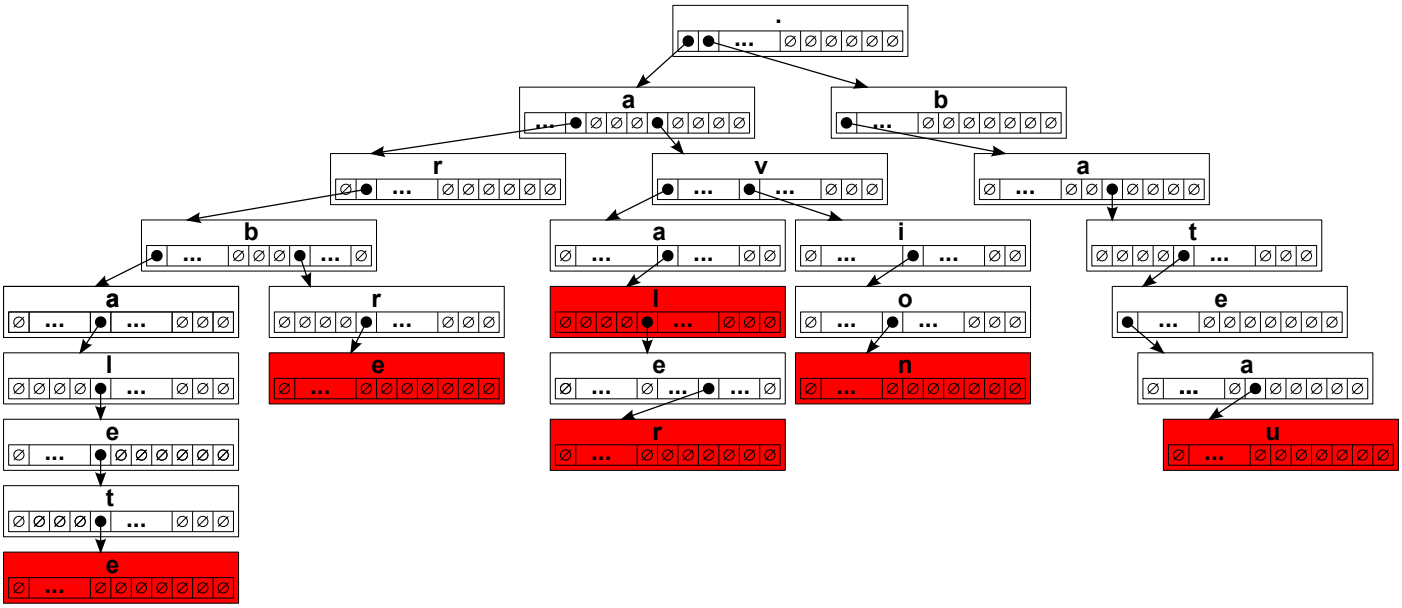


FIGURE 3 – Arbre lexicographique représenté avec des noeuds contenant un tableau de pointeurs sur les noeuds fils. Le symbole ∅ indique que la case correspondante du tableau contient la valeur `NULL`.

```
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 ?

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 5
Bibliothèque générique de liste

Version du 27 août 2024

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`).

TD Séance 6
Utilisation de la (GNU) libC

Version du 27 août 2024

Objectif(s)

✯ Apprendre à utiliser une bibliothèque générique standardisée

Exercice(s)

La libC, la bibliothèque standard de fonctions du langage C contient des fonctions pour rechercher des valeurs ou trier des ensembles de valeurs. Certaines de ces fonctions permettent de manipuler directement des tableaux, d'autres s'appuient sur des arbres. Nous allons voir comment les utiliser.

Exercice 1 *(entraînement)* – Tableaux

La libC inclut plusieurs fonctions de recherche dans un tableau (fichier header à inclure : `search.h`) :

```
void * lfind (const void *key, const void *array, size_t *nmemb, size_t size, comparison_fn_t compar)
void * lsearch (const void *key, void *array, size_t *nmemb, size_t size, comparison_fn_t compar)
void * bsearch (const void *key, const void *array, size_t nmemb, size_t size, comparison_fn_t compare)
```

`lfind` recherche une valeur dans un tableau non trié, `lsearch` fait la même chose, mais ajoute l'élément recherché à la fin du tableau s'il n'y figure pas déjà (la mémoire allouée au tableau doit le permettre) et `bsearch` fait une recherche par dichotomie sur un tableau trié (sans ajout, le 'b' est pour *binary*).

Le principe de ces différentes fonctions est de transmettre la valeur à chercher sous la forme d'un pointeur générique : `void *key`. Le `const` indique que ces différentes fonctions ne vont pas modifier cette valeur.

`array` est le tableau dans lequel on souhaite chercher (et éventuellement ajouter) les valeurs. Il contient `nmemb` éléments qui sont tous de taille `size` octets. Attention, pour `lfind` et `lsearch`, il est donné par pointeur. Cela permet à `lsearch` d'incrémenter cette valeur si l'élément recherché est ajouté à la fin du tableau [1].

`compar` est une fonction permettant de comparer deux éléments. Son prototype est le suivant :

```
int compar(const void *v1, const void *v2);
```

La fonction transmise en argument doit permettre de comparer deux éléments et de renvoyer une valeur positive, nulle ou négative si la donnée pointée par `v1` est, respectivement, supérieur, égal ou inférieur à celle pointée par `v2`.

1. Rappelez la fonction de comparaison dans le cas où les données à comparer sont des entiers.
   Soit un tableau d'entiers `tab` de taille 100 :
   ```
   int tab[100];
   ```
2. Comment utiliser `lfind` pour trouver si la valeur 42 est dans ce tableau ? Ecrivez les instructions permettant d'afficher un message si cette valeur est trouvée.

1. Il n'était pas nécessaire de transmettre `nmemb` par pointeur pour `lfind`, mais ce choix a probablement été fait pour que les deux fonctions aient, quasiment, le même prototype.

3. À l'aide de la fonction `lsearch`, écrivez les instructions permettant de créer un tableau contenant des valeurs aléatoires toutes différentes. Le tableau sera de taille `nb_max`, vous l'allouerez dynamiquement.

   La fonction de recherche `bsearch` est bien plus rapide que `lfind` et `lsearch` car elle s'appuie sur le fait que le tableau est trié (elle procède par dichotomie). Il faut donc trier le tableau avant d'utiliser cette fonction.

4. Utilisez la fonction `qsort` pour trier le tableau. Pour rappel, le prototype de cette fonction est le suivant :

   ```
   void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));
   ```

5. En supposant le tableau `tab` trié, écrivez la ou les instructions nécessaires pour rechercher la valeur 42 dans le tableau avec la fonction `bsearch`.

Exercice 2 *(approfondissement)* – Arbres

La libC inclut des fonctions pour implémenter des arbres binaires de recherche. L'interface est conçue pour être minimaliste. Tout passe par 4 fonctions dont le prototype est déclaré dans `search.h` :

```
void *tdelete(const void * key, void ** rootp, int (*compar) (const void *key1, const void *key2));
void *tfind(const void *key, void *const *rootp, int (*compar) (const void *key1, const void *key2));
void *tsearch(const void *key, void **rootp, int (*compar) (const void *key1, const void *key2));
void twalk(const void *root, void (*action) (const void *node, VISIT order, int level));
```

à laquelle la GNU libC ajoute :

```
void tdestroy(void *root, void (*free_node)(void *nodep));
```

(dans la libC non GNU, la libération de l'arbre doit se faire en libérant tous les noeuds avec une boucle).

Comme pour les listes, `tfind` et `tsearch` cherchent une valeur dans l'arbre. `tsearch` l'ajoute dans l'arbre si elle n'y figure pas. C'est la fonction à utiliser pour construire l'arbre. `twalk` permet d'appliquer une action à tous les noeuds de l'arbre et `tdelete` permet de détruire un noeud.

La structure utilisée pour les noeuds n'est pas exposée. En fait, on n'a pas besoin de la connaître pour utiliser ces fonctions. Cela s'appuie sur une astuce liée au fait que le champ de donnée est en première position dans la structure du noeud :

```
typedef struct node_t
{
  const void *key;
   /* autres champs (pointeurs sur les sous-arbres et autres) */
} *node;
```

Dans la mémoire allouée à un noeud, le champ `key` (la donnée), est placé en premier, donc un pointeur sur un noeud pointe également sur la donnée ! Il suffit de le transformer dans le bon type pour ne récupérer que les octets de la donnée et non ceux de tout le noeud. Ce n'est pas très intuitif, mais c'est astucieux !

Cela signifie que si on dispose d'un pointeur `pn` sur un noeud, on dispose d'un pointeur vers le champ `key`. Il s'agit d'un pointeur sur un pointeur sur la donnée que porte le noeud. Il suffit de faire les bonnes transformations de types pour récupérer la donnée. Au final, on n'a donc pas besoin de connaître le type du noeud pour utiliser ces fonctions.

Exemple :

```
node n;
int v=42;
n.key=&v;
node *pn=&n;
int **ppv=&n.key;
printf("Adresse de n : %p, adresse du champ key de n: %p\n",pn,ppv);

printf("Valeur du champ key: %d\n", **ppv);
// Utilisation du pointeur sur le noeud pour acceder a la donnee:
int **ppv2=(int**)&n;
printf("Valeur du champ key en passant par ppv2: %d\n",**ppv2);

// on pourrait n'acceder au pointeur sur le noeud que via un pointeur generique:
void *ng=(void *)pn;
int **ppv3=(int**)&ng;
printf("Valeur du champ key en passant par ppv3: %d\n",**ppv3);
```

Exemple d'affichage :

```
Adresse de n : 0x7ffeef816870, adresse du champ key de n: 0x7ffeef816870
Valeur du champ key: 42
Valeur du champ key en passant par ppv2: 42
Valeur du champ key en passant par ppv3: 42
```

La fonction d'ajout dans l'arbre s'appuie sur des algorithmes permettant de garder l'arbre aussi équilibré que possible pour que la recherche soit efficace quel que soit l'ordre d'ajout des éléments.

Des explications plus détaillées sur les fonctions sont données dans la suite de l'énoncé.

La fonction `tsearch` prend en argument la donnée cherchée (`const void *key`), la racine de l'arbre (`void **rootp`) et la fonction permettant de comparer des valeurs.

En cas d'ajout d'une valeur à l'arbre, la racine peut être modifiée, `tsearch` a donc besoin de pouvoir modifier cette variable. C'est la raison pour laquelle la racine de l'arbre est transmise sous la forme d'un double pointeur (il s'agit donc d'un pointeur transmis par pointeur pour pouvoir le modifier...).

Attention : la donnée n'est pas recopiée s'il y a un ajout. Cela signifie qu'elle doit être allouée par ailleurs et rester accessible tant que l'on souhaite utiliser l'arbre.

1. Écrivez les instructions permettant de créer un arbre contenant `nb_max` valeurs tirées aléatoirement et toutes différentes. Vous prendrez soin d'éviter toute fuite de mémoire.

   La fonction de recherche `tfind` prend en argument la donnée à chercher (`const void *key`), le pointeur sur la racine et la fonction de comparaison. Le pointeur sur la racine est déclaré de la façon suivante :
   `void *const *rootp`. Le const indique que l'on ne peut pas modifier ce sur quoi pointe `rootp`.

   Remarque sur const et pointeurs simples et doubles : dans le cas d'un pointeur simple dont on veut indiquer que la donnée n'est pas modifiable, on écrit `const int *pi`. On pourrait aussi écrire `int const *pi`, ce qui se lit de droite à gauche : pointeur (*) sur constant (`const`) entier (`int`), ce qui est plus grammaticalement correct en Anglais : pointer to a const int... Cela permet de mieux comprendre la déclaration de `rootp` : il s'agit d'un pointeur sur une variable constant de type XX, le XX étant ici un void *. C'est donc un pointeur sur un pointeur générique constant, CQFD.
2. Écrivez la ou les instructions permettant de chercher la valeur 42 dans l'arbre.

   La fonction `twalk` permet d'appliquer une fonction donnée à chacun des noeuds de l'arbre avec un parcours en profondeur, main gauche. Le premier argument est la racine de l'arbre (const car la fonction ne peut la modifier) et le second est l'action à appliquer à chaque noeud. Cette fonction doit respecter le prototype suivant :

   ```
   void action (const void *node, VISIT order, int level)
   ```

   Elle ne renvoie donc rien et prend trois arguments. `node` est un pointeur sur le noeud courant (qui ne peut être modifié). `order` est en entier indiquant le moment de la "visite". En parcourant un arbre binaire, on peut tomber plusieurs fois sur un noeud selon si on le regarde avant ses sous-arbres, entre les deux sous-arbres ou après les sous-arbres. `VISIT` est un type énuméré défini de la façon suivante pour déterminer à quel moment l'action a été appelée :

   ```
   typedef enum { preorder, postorder, endorder, leaf } VISIT;
   ```

   `preorder` indique que, pour cet appel de l'action, les sous-arbres n'ont pas encore été considérés, `postorder` que le premier a déjà été considéré et `endorder` que les deux ont été considérés. `leaf` indique que le noeud est une feuille (et n'a donc pas de sous-arbres).

   `level` indique le niveau du noeud sur lequel on applique l'action (0 pour la racine).
3. Utilisez la fonction `twalk` pour afficher le contenu du tableau. Vous écrirez les trois types d'affichage (préfixe, infixe et postfixe) qui seront ensuite utilisés en tant que paramètre de twalk.

   La fonction `tdestroy` parcourt l'arbre, libère tous les noeuds et libère également les données en appelant la fonction `free_node` passée en 2eme argument. La fonction `tdelete` libère le noeud contenant la donnée passée dans l'argument `key`. Comme pour `tsearch`, la racine de l'arbre est passée en double pointeur car cela peut être elle qui est détruite. Comme précédemment, la fonction `compar` permet de comparer des données.

4. Écrivez les instructions pour libérer l'arbre avec soit `tdestroy`, soit `tdelete`. Avec `tdestroy`, la destruction est réalisée en une seule instruction. Avec `tdelete`, il faut faire une boucle dans laquelle, par exemple, on demande systématiquement de détruire la racine (donc le noeud qui contient la même valeur que la racine...).
5. En pratique, `lfind` est la méthode la plus lente. `tsearch` bien plus rapide et `bsearch` est encore plus rapide. Est-ce normal ? Pourquoi avoir gardé `tsearch` plutôt que de se contenter de `bsearch` ?

Exercice 3 *(approfondissement)* – Autres types de données

1. Que faut-il changer au code pour utiliser d'autres types de données ?