SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– LU2IN018

Examen du juin 2020
1h30
Aucun document n'est autorisé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs. Le barème sur 60 points (10 questions) n'a qu'une valeur indicative.

Les appels à `malloc` seront supposés réussir. Il ne sera pas nécessaire de vérifier leur valeur de retour. De la même manière, l'ouverture d'un fichier sera supposée réussir. Il ne sera pas nécessaire de vérifier que c'est bien le cas.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé.

L'ensemble des structures et prototypes de fonctions est également rappelé à la fin de l'énoncé, sur une page détachable.

Détection de gènes

L'objectif de ce projet est de détecter des gènes dans des génomes bactériens. Un génome (souvent appelé ADN) est une grosse molécule composées de millions de plus petites molécules nommées nucléotides. Il n'existe dans l'ADN que 4 nucléotides différents : A, T, G, C. Cette suite de millions de lettres forme en informatique un très long texte dans lequel il faut identifier les gènes qui sont des morceaux de ce texte.
Un génome sera donc stocké sous la forme d'une chaine de caractères. Nous écrirons ici les fonctions pour identifier les gènes que nous stockerons dans une liste chainée générique simplement chaînée.

1 Détection rapide des codons *start* et *stop*

L'ADN est un texte qui se lit 3 lettres par 3 lettres. Ces successions de 3 lettres sont nommées codon. Un gène commence par un codon *start* "ATG" et se termine par un codon *stop* ("TAA" ou "TAG" ou "TGA") **en phase**, c'est-à-dire qu'il faut que la longueur totale du gène soit un multiple de 3. Voici un exemple de gène où les codons sont volontairement espacés pour mieux les visualiser (il n'y a par contre pas d'espace dans les chaînes de caractères à traiter) :
ATG TTA CCT CAC ACA TAC ACA GAT AAC **TAA**

Question 1 (3 points)
Ecrivez la fonction

```
int Est_Start(char *seq);
```

qui retourne 1 si le codon donné en argument commence par un codon *start* "ATG" et 0 sinon.

Réponse :

Question 2 (6 points)

Il existe 3 codons *stop*. Pour accélérer leur détection, nous allons les stocker dans un arbre lexicographique dont voici la structure pour un nœud :

```
typedef struct _tyArbre{
  struct _tyArbre *tFils[26];
} tyArbre;
```

Un nœud contient donc un tableau de 26 pointeurs vers d'autre nœuds fils (à NULL s'il n'y a pas de fils pour cette lettre). Le tableau comporte 26 cases même s'il n'y a que 4 nucléotides différents pour faciliter le calcul des indices des lettres dans ce tableau. Lorsqu'une lettre est présente dans l'arbre, son pointeur n'est pas NULL. L'arbre lexicographique pour les 3 codons stop est présenté figure 1.

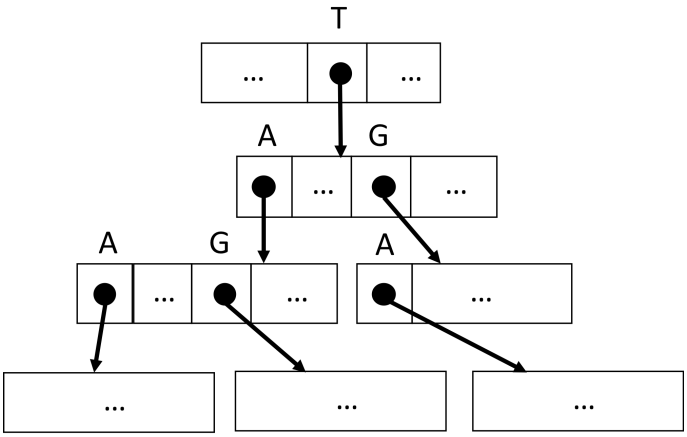

FIGURE 1 – Le tableau tFils d'un nœud a toujours 26 cases mais elles ne sont pas représentées si elles contiennent NULL. Dans ce cas, elles sont représentées par "...". Voici comment le lire : le codon "TAA" est présent car : à la racine, la case "T" contient un pointeur valide vers un tableau qui lui même contient un pointeur valide à la case "A" vers un tableau qui lui même contient un pointeur valide à la case "A" qui lui aussi contient un pointeur vers un tableau qui lui ne contient que des pointeurs NULL.

Ecrire la fonction qui permet d'ajouter un codon dans l'arbre de prototype :

```
tyArbre *Ajouter_Codon(tyArbre *pRacine, char *codon);
```

Elle prend en argument pRacine la racine de l'arbre, qui peut ou non être NULL et codon, un codon qui est une chaine de caractères de 3 lettres.

Réponse :

Question 3 (6 points)

Ecrire maintenant la fonction de prototype

```
tyArbre *Creer_Arbre_Codons_Stop();
```

qui ne prend aucun argument et retourne l'arbre construit avec les codons *stop* ("TAA", "TAG" et "TGA").

Réponse :

Question 4 (6 points)

Ecrire maintenant la fonction de prototype

```
int Est_Stop(tyArbre *pRacine, char *codon);
```

qui prend en argument l'arbre construit avec les codons *stop* et un codon et qui retourne 1 si le codon donné en argument est un codon *stop* et 0 sinon.

Réponse :

Question 5 (6 points)

Ecrire maintenant de prototype la fonction de libération de l'arbre :

```
void Liberer_Arbre(tyArbre *pNoeud);
```

.

Réponse :

2 Identification des gènes

Un gène sera stocké dans une structure dont voici la définition :

```
typedef struct {
  int start; /*indice du premier nucleotide du premier codon (start)
     pour ce gene*/
  int stop; /*indice du premier nucleotide du dernier codon (stop)
     inclus dans ce gene*/
  char *pSeq; /*Pointeur vers le genome (debut du genome et non du
     gene)*/
} tyGene;
```

L'ensemble des gènes sera stocké dans une liste chaînée générique dont les structures et les prototypes des fonctions sont donnés ci-dessous. Les données (champ `data`) seront donc ici de type `tyGene`.

```
typedef struct _element {
  void *data;
  struct _element * suivant;
} Element;

typedef struct _liste {
  Element * elements;
  void (*detruire)(void *data);
  void (*afficher)( void *data);
} Liste;

// insere en debut de liste
void Inserer_Debut(Liste * pliste, void *data);

// detruire une liste
void Detruire_Liste(Liste * pliste);

// afficher une liste
void Afficher_Liste(Liste * pliste);
```

Question 6 (6 points)

Écrivez une fonction permettant d'allouer la mémoire pour une structure de type `tyGene` et d'initialiser ses différents champs avec les valeurs données en arguments. Prototype :

```
tyGene* New_Gene(int start, int stop,  char *pS);
```

Réponse :

Question 7 (6 points)

La liste étant générique, il faut lui fournir les pointeurs vers des fonctions permettant de manipuler les données. Nous avons besoin ici d'une fonction pour libérer les données que nous nommerons `Free_Gene`, dont le pointeur est dans le champ `detruire`, et une fonction pour afficher les données que nous nommerons `Afficher_Gene`, dont le pointeur est dans le champ `afficher`

Écrivez ces deux fonctions. Vous déterminerez avec soin leurs prototypes.

L'affichage d'un gène devra ressembler à ceci :

```
> 387-396 12
ATGATGCATTAG
```

avec sur la ligne commençant par ">", l'indice dans le génome du codon *start* dans le génome, puis l'indice du codon *stop* (indice du premier nucléotide du codon), puis la longueur du gène, puis sur la ligne suivante, les nucléotides entre le codon *start* et le codon *stop* (inclus), sur une seule ligne.

Réponse :

Question 8 (6 points)

Écrivez la fonction permettant d'insérer en tête dans la liste. Les données ne seront pas dupliquées. Prototype :

```
void Inserer_Debut(Liste * pliste, void *data);
```

Réponse :

Bien qu'on ne l'écrive pas, on supposera dans les questions suivantes que les fonctions `void afficher_liste (Liste * pliste);` et `Detruire_Liste(Liste * pliste);` existent et fonctionnent.

Question 9 (9 points)

Vous allez maintenant écrire la fonction permettant d'identifier les gènes potentiels, délimités par un codon *start* et un codon *stop* en phase. Prototype :

```
Liste *Trouver_Gene(char *pS, tyArbre *arbre_stop);
```

Il faut donc chercher les gènes potentiels dans les 3 "phases de lecture", c'est-à-dire en commençant la recherche au 1er nucléotide puis en sautant de 3 en 3, mais aussi en commençant au 2nd nucléotide et au 3e nucléotide. Par exemple, le fragment de génome suivant a un gène entre les codons soulignés dans la 1ere phase, mais il a un autre gène entre les codons en gras dans la 3e phase.

ATG GAT AT**A TG**G TTT TTA AGA TAA ACA TGT GT**T AA**G

Cette fonction retournera une liste que vous aurez allouée et dont vous aurez correctement initialisé tous les champs. Il est conseillé de ne parcourir le génome qu'une seule fois ; vous serez pénalisé si vous le parcourez plusieurs fois. Vous utiliserez les fonctions `Est_Start` et `Est_Stop` pour identifier les codons.

Réponse :

Question 10 (6 points)

Ecrivez la fonction `main` où l'arbre des codons *stop* sera construit, un fichier nommé `Buchnera.fasta` sera lu, les gènes seront ensuite identifiés puis filtrés puis affichés. Vous n'oublierez pas de libérer toute la mémoire allouée. Pour lire le génome, vous disposez de la fonction

```
char *Lire_Fasta(char *nomFi);
```

lisant le génome contenu dans le fichier de nom `nomFi` et retournant une chaine de caractères contenant le génome.

Réponse :

Mémento de l'UE 2I001

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

```
int printf(const char *format, ...);
```

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
```

La fonction `scanf` permet de saisir et analyser un texte saisi sur l'entrée standard (par défaut le clavier). Le texte saisi devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion son identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie **EOF** en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code **NULL** sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code **EOF** sinon `0`.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données lues sont stockées en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Écriture de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données à écrire sont lues en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans `string.h`.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin `\0`.

```
size_t strlen (const char *s);
```

Renvoie la longueur d'une chaîne de caractères (marqueur de fin `\0` non compris).

```
int strcmp (const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux `n` premiers caractères. La valeur retournée est :

- 0 si les deux chaînes sont identiques,
- négative si `s1` précède `s2` dans l'ordre lexicographique (généralisation de l'ordre alphabétique),
- positive sinon.

```
char *strcpy (char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne `src` dans la chaîne `dest` (marqueur de fin `\0` compris). La chaîne `dest` doit avoir précédemment été allouée. La copie peut être limitée à `n` caractères et la valeur retournée correspond au pointeur de destination `dest`.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie `n` octets à partir de l'adresse contenue dans le pointeur `src` vers l'adresse stockée dans `dest`. `dest` doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. `memcpy` renvoie la valeur de `dest`.

```
size_t strlen(const char *s);
```

Retourne le nombre de caractères de la chaîne `s` (marqueur de fin `\0` non compris).

```
char *strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction `free`.

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ajoute la chaîne `src` à la suite de la chaîne `dst`. La chaîne `dest` devra avoir été allouée et être de taille suffisante. La fonction retourne `dest`.

```
char *strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne `needle` rencontrée dans la chaîne `haystack`. Si la chaîne recherchée n'est pas présente, la fonction retourne `NULL`.

Conversion de chaînes de caractères

Prototypes disponibles dans `stdlib.h`.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par `nptr` en un entier de type `int`.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par `nptr` en un `double`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne `nptr` en un entier long. l'interprétation tient compte de la `base` et la variable pointée par `endptr` est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans `stdlib.h`.

```
void *malloc(size_t size);
```

Alloue `size` octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie `NULL` en cas d'échec.

```
void *realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`. `size` correspond à la taille en octet de la nouvelle zone allouée. `realloc` garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`.

La liste des fonctions du programme considéré est indiquée ci-après. Certaines fonctions ne sont pas à écrire. Ces fonctions peuvent tout de même être utilisées et considérées comme disponibles.

```
typedef struct _tyArbre{
  struct _tyArbre *tFils[26];
} tyArbre;


char *Lire_Fasta(char *nomFi);
int Est_Start(char *seq);
tyArbre *Ajouter_Codon(tyArbre *pRacine, char *codon);
int Est_Stop(tyArbre *pRacine, char *codon);
void Liberer_Arbre(tyArbre *pNoeud);
tyArbre *Creer_Arbre_Codons_Stop();

typedef struct {
  int start; /*indice du premier nucleotide du premier codon (start)
     pour ce gene*/
  int stop; /*indice du premier nucleotide du dernier codon (stop)
     inclus dans ce gene*/
  char *pSeq; /*Pointeur vers le genome (debut du genome et non du gene
     )*/
} tyGene;



void Afficher_Gene(void *pData);
tyGene* New_Gene(int start, int stop,  char *pS);
void Free_Gene(void *pData);
Liste *Trouver_Gene(char *pS, tyArbre *arbre_stop);
int Est_Trop_Court(void *pData, void *oa);

typedef struct _element {
  void *data;
  struct _element * suivant;
} Element;

typedef struct _liste {
  Element * elements;
  void (*detruire)(void *data);
  void (*afficher)( void *data);
} Liste;

// insere en debut de liste
void Inserer_Debut(Liste * pliste, void *data);

// detruire une liste
void Detruire_Liste(Liste * pliste);
```

```
// afficher une liste
void Afficher_Liste(Liste * pliste);
```