Algorithmique 2 : structures de données linéaires
Fiche de TP n° 0

Travaux pratiques : modalités et réglages

Environnement

Lors des séances de TP, vous travaillerez en priorité sur les machines mises à votre disposition par le département d'informatique et sous Xubuntu. Si vous souhaitez travailler sur votre propre machine, vous le ferez à vos risques et périls et devrez vous débrouiller.

Modalités

À la fin de chaque séance, vous devez impérativement déposer dans le cours Algorithmique 2 de la plateforme UniversiTICE une archive au format `.tar.gz` du dossier contenant, à sa racine ou dans ses sous-dossiers, les fichiers sources `.c`, les fichiers sources `.h` et les fichiers `makefile`, et uniquement ceux-là, fournis, créés, modifiés ou utilisés à l'occasion de la ou des séances dévolues au traitement des exercices figurant sur la fiche. Vous pourrez par la suite déposer des versions améliorées.

Un fichier `makefile` est fourni avec le dossier à travailler qui permet la création de l'archive aux format et contenu idoines via les commandes « `make` » ou « `make dist` ».

Les enseignant·es pourront refuser de prendre en compte tout source qui n'aura pas été remis en forme par l'utilitaire de remise en forme de codes sources Uncrustify.

Réglages de l'EDI Geany en mode copie

Avant de lancer Geany, récupérez l'archive `algo2_src.tar.gz` sur UniversiTICE, extrayez-en ses composants puis copiez dans votre dossier personnel `~/` le contenu du dossier `config/` [1].

Réglages partiels de l'EDI Geany en mode manuel

Les « préférences de l'utilisateur » décrites par le fichier de configuration `geany.conf` peuvent être effectuées à la main dans trois boites de dialogue.

1) Dans la boite de dialogue « Éditer > Format > Envoyer la sélection vers > Définir les commandes personnalisées » : fixer le champ « Commande » de l'article dont le champ « ID » vaut 1 à « `bash -c "uncrustify -c $HOME/.uncrustify072_c.cfg -lc -q"` ». Si cet article n'existe pas déjà : « Ajouter ».

2) Dans la boite de dialogue « Éditer > Préférences » :
Général > Divers > Divers : éteindre « Émettre un bip sur les erreurs ou lorsque la compilation est terminée » ;

1. Le dossier `config/` est composé de cinq fichiers et trois sous-dossiers. Trois fichiers de configuration de l'EDI Geany : `geany.conf`, `keybindings.conf` et `filetypes.c`, un fichier configuration pour Uncrustify : `.uncrustify072_c.cfg`, un script : `config.sh`. Les deux premiers, à trouver dans `config/.config/geany/`, sont à copier dans `~/.config/geany/`, le troisième, à trouver dans `config/.config/geany/filedefs/`, dans `~/.config/geany/filedefs/`, le quatrième, à trouver dans `config/`, dans `~/`, le cinquième, lui aussi dans `config/`, peut être copié dans `~/` ou ne pas être copié du tout. Attention : les dossiers et fichiers dont les noms commencent par un point sont cachés par défaut dans les fenêtres gestionnaire de fichiers ; si vous voulez qu'ils y soient affichés, « Affichage > Afficher les fichiers cachés » ou tapez ctrl+h. Pour réaliser la copie attendue très simplement : ouvrez un terminal dans le dossier `config/` et exécutez le script `config.sh` : « `./config.sh` » en ligne de commande. Si la configuration s'est correctement déroulée, vous verrez apparaitre un filet vertical rouge en colonne 80 à l'ouverture de Geany en lieu et place du filet vert, à peine visible, placé en colonne 72 et proposé par défaut.

Interface > Interface > Police > Éditeur : la fixer à « Liberation Mono Regular 10 » ;
Éditer > Fonctionnalités > Fonctionnalités : fixer « Marqueur de commentaire » à la chaine vide ;
Éditer > Indentation > Indentation : fixer « Largeur » à 2 ;
Éditer > Indentation > Indentation : fixer « Type » à « Espaces » ;
Éditer > Complétions > Complétions : allumer « Compléter automatiquement tous les mots du document » ;
Éditer > Complétions > Complétions : fixer « Caractères à taper... » à 3 ;
Éditer > Affichage > Affichage : allumer « Afficher les guides d'indentation » ;
Éditer > Affichage > Affichage : allumer « Afficher les espaces » ;
Éditer > Affichage > Marqueur de longues lignes : fixer « Colonne » à 80 ;
Éditer > Affichage > Marqueur de longues lignes : fixer « Couleur » au rouge, soit « Couleur > Personnalisée > + > `#FF0000` > Sélectionner » ;
Fichiers > Enregistrement des fichiers : allumer « Enlever les espaces et tabulations de fin ».

3) Dans la boite de dialogue « Affichage » :
Changer le jeu de couleurs... : choisir « Alternate » ;
Afficher la barre d'outils : éteindre.

Greffons Geany intéressants

Parmi les quelques greffons (*plugins*) dont dispose Geany, trois au moins peuvent être ajoutés qui peuvent améliorer sensiblement la frappe de code ou travailler à sa conformité. Ils sont activables ou désactivables à partir de « Outils > Gestionnaire de plugin ».

Auto-close Ferme automatiquement les parenthèses, les accolades, les crochets...

Define formatter Aligne automatiquement sur la colonne 79 (un de moins que la valeur de la colonne de marqueur de longues lignes) les barres obliques inverses qui permettent de prolonger la définition des macros sur la ligne du dessous. Il est somme toute souvent nécessaire de retravailler un peu le code après la remise en forme par Uncrustify.

Vérification orthographique Vérifie l'orthographe des commentaires et des chaines de caractères constantes.

Certains greffons peuvent être paramétrés. C'est par exemple le cas du premier et du troisième de ceux cités à l'instant. Vous pouvez paramétrer un greffon immédiatement après avoir allumé sa demande d'activation. Vous pouvez aussi, si au moins un greffon est activé, passer par « Éditer > Préférences des plugins ».

Pour « Auto-close », il est suggéré de désactiver « Guillemets simples '' ».

Pour « Vérification orthographique », il est suggéré d'activer au minimum « Vérifier l'orthographe au cours de la frappe ».

Raccourcis de l'EDI Geany

Voici ce que permettent certaines touches ou combinaisons de touches.

f8 : vérifier la syntaxe du fichier d'extension `.c` ou `.h` en cours d'édition avec les options de compilation standards. En cas de succès pour un fichier `.c`, produire le fichier objet associé d'extension `.o`. En cas d'échec ou de succès pour un fichier `.h`, produire le fichier en-tête pré-compilé (*precompiled header file*) associé d'extension `.gch`[2].

f9 : construire l'exécutable associé au fichier d'extension `.c` en cours d'édition avec les options de compilation standards. Ne peut être utilisée que pour des fichiers complets, hors compilation séparée.

2. Il vous appartient de supprimer vous-même les éventuels fichiers `.gch` qui figurent dans votre dossier avant que de créer l'archive à déposer.

maj+f9 : construire l'exécutable décrit dans le fichier `makefile` qui figure dans le dossier du fichier d'extension `.c` ou `.h` ou encore du fichier `makefile` en cours d'édition. Le même résultat peut être obtenu en ligne de commande par « `make` » ou « `make all` ».
maj+ctrl+f9 – « clean » +entrée : même résultat que « `make clean` ».
ctrl+alt+a : remettre en forme par Uncrustify [3] de la partie du source C sélectionnée.
ctrl+a – ctrl+alt+a : sélectionner tout le texte en cours d'édition (première combinaison) puis le remettre en forme par Uncrustify (deuxième). À n'utiliser que sur des fichiers sources C.
alt+d – i : supprimer les indicateurs d'erreurs (dont les tôles ondulées générées par le correcteur orthographique).

Suite des réglages

Sur Xubuntu, et donc en particulier dans les salles de TP du département d'informatique, la configuration peut être prolongée comme suit.

Cliquez droit sur le premier fichier d'extension `.c` venu. Dans « Propriétés > Général > Ouvrir avec », signifiez que vous l'ouvrirez dorénavant — lui comme tous ceux de même extension — avec Geany. Vous ferez par la suite de même avec le premier fichier d'extension `.h` et le premier fichier `makefile` que vous rencontrerez.

Make sous Geany

Geany cherche par défaut à lancer le fichier `makefile` qui se situe dans le même dossier que le fichier en cours d'édition.

Dans le cadre du développement ou de la mise au point d'un ou de plusieurs modules situés dans des dossiers distincts du dossier dans lequel figurent le source contenant la fonction principale `main` et le fichier `makefile`, il faut activer le source principal ou le fichier `makefile` avant de lancer — via maj+f9 — la construction. Ce qui peut vite s'avérer insupportable. Une solution consiste à personnaliser, le temps desdits développement ou mise au point, certaines des commandes de construction « indépendantes » liées à `make`.

Soit *chemin* le chemin (absolu, ou alors relatif si tous les fichiers sont au même niveau) du dossier dans lequel figurent le source contenant la fonction principale `main` et le fichier `makefile`. Dans la boite de dialogue « Construire > Définir les commandes de construction » :
Commandes indépendantes : fixer le champ « Commande » de l'article dont le champ « Étiquette » vaut « Make » à « `(cd` *chemin* `&& make)` » ;
Commandes indépendantes : fixer le champ « Commande » de l'article dont le champ « Étiquette » vaut « Make Custom Target... » à « `(cd` *chemin* `&& make clean)` ».

Par la suite, et quel que soit le source actif :
maj+f9 : pour « `make` » ou « `make all` » ;
maj+ctrl+f9 – entrée : pour « `make clean` ».

3. La remise en forme est intéressante mais n'est pas parfaite. De nouvelles versions du fichier de configuration `.uncrustify072_c.cfg` pourront être fournies dans le courant du semestre.

Algorithmique 2 : structures de données linéaires
Fiche de TP n° 1

Programmation récursive d'opérations sur les naturels

Objectifs : premiers codages de fonctions récursives.
Prérequis : avoir travaillé les sections 2.1 à 2.4 du support de cours et d'exercices ; avoir abordé l'exercice 2.10 dudit support.
Travail minimum : exercice 1.

Récupérez le dossier `algo2_src/tp/1/` qui figure dans l'archive `algo2_src.tar.gz` sur UniversiTICE. Prenez connaissance du contenu des fichiers en-têtes `"nat.h"` et `"natio.h"` à trouver respectivement dans les sous-dossiers `nat/` et `natio/`.

Placez-vous dans le sous-dossier `natio/`. Prenez connaissance du contenu du fichier source `natio_test.c`. Construisez [4] l'exécutable associé. Testez quelques entrées, comme par exemple [5] :

```
$ cat > test.txt
0 0
sss0 sssss0
  ps0s0pp0 pss0
  ss0 sssssssssssssssssssssssssssssssssssssssssssssssssss0
$ ./natio_test < test.txt
0
0
sss0
sssss0
0
s0
UNDEF
s0
ss0
INFTY
```

Seule une partie du module `natio` a été implantée, à savoir les fonctions `input` et `output`, lesquelles ne mettent à disposition qu'un format externe des naturels des plus rudimentaires. L'objet de l'exercice 2 est de remédier à ce problème. En attendant, c'est avec ce format fait de `0`, `s` et `p` qu'il vous faut envisager toute exécution du précédent exécutable et de votre solution à l'exercice 1.

Un conseil : afin de profiter de la coloration syntaxique et du rappel des prototypes lors de la saisie fournis par Geany, gardez les en-têtes `"nat.h"` et `"natio.h"` ouverts.

Exercice 1

Placez-vous dans le sous-dossier `natop/`, puis prenez connaissance du contenu de l'en-tête `"natop.h"` ainsi que de celui des sources `natop.c` et `natop_test.c`.

Construisez. Testez. Par exemple :

4. Dorénavant, tous les exécutables à produire sont issus de compilations séparées. Si, ici, vous obtenez comme message d'erreur « `natio_test.c:3:10: fatal error: nat.h: Aucun fichier ou dossier de ce type` », vous ne devez pas chercher à déplacer ou dupliquer l'en-tête `"nat.h"` dans le dossier courant : vous devez taper maj+f9 sous Geany ou « `make` » en ligne de commande.

5. Si vous suivez cette suggestion, pensez à supprimer le fichier temporaire `test.txt` avant de confectionner l'archive à rendre.

```
$ ./natop_test < ../natio/test.txt
sum(0, 0) = 0
sum(sss0, sssss0) = ssssssss0
sum(0, s0) = s0
sum(UNDEF, s0) = UNDEF
sum(ss0, INFTY) = INFTY
```

Programmez en C toutes les opérations de l'exercice 2.10 — en plus de la somme donc — dans le cadre plus complexe fixé par le module `nat` ; inscrivez leurs codes dans le source `natop.c`. Modifiez de conserve le contenu du source `natop_test.c` de telle sorte que, pour tous les couples donnés en entrée, le résultat de toutes les opérations implantées soit affiché.

Chaque implantation doit être récursive. Chaque fonction récursive ne doit faire appel qu'à elle-même ou aux fonctions du module `nat`. Vous ne modifierez en aucun cas le module `nat`, ni — à ce stade — le module `natio`, ni l'en-tête `"natop.h"`.

Exercice 2

Complétez le module `natio` en donnant corps aux fonctions `dinput` et `douput` ; de ce module, vous ne devez modifier que la partie implantation `natio.c` et en aucun cas la partie interface `"natio.h"`. Recourrez aux fonctions standards `fscanf` et `fprintf`.

Testez tout d'abord dans le sous-dossier `natio/` à l'aide du source `test.c`. Si vous avez implanté `douput` par exemple, remplacez dans `test.c` l'appel à `output` par un appel à `douput`. Vous obtiendrez alors :

```
$ ./natio_test < test.txt
0
0
3
5
0
1
UNDEF
1
2
INFTY
```

Testez ensuite dans le sous-dossier `natop/`.

Exercice 3

Complétez le module `natop` en implantant les opérations qui ne figurent pas dans l'exercice 2.10 : produit ; quotient et reste de la division euclidienne. Une contrainte est relâchée : les fonctions préalablement définies dans le module à l'exercice 1 peuvent être utilisées. Testez.

L'implantation par défaut du type `nat` ne permet qu'une gestion de valeurs inférieures à quelques dizaines. Testez les deux programmes précédents — tests sur les entrées et les sorties et sur les opérations — avec la deuxième implantation proposée, assurément beaucoup moins limitée : ajoutez pour cela « `-DNAT_IMPLEMENTATION=HIGH` » à la variable `CFLAGS` des fichiers `makefile` puis construisez.

Exercice 4

Dans la mesure du possible — une fois au moins abordé le chapitre 3 du support de cours et d'exercices —, efforcez-vous de donner de toutes vos fonctions récursives des versions exclusivement récursives terminales.

Algorithmique 2 : structures de données linéaires
Fiche de TP n° 2

Programmation récursive de fractales

Objectifs : codage d'opérations récursives d'ordre de récursion multiple.
Prérequis : géométrie dans le plan niveau collège-lycée.
Travail minimum : exercice 1.

Récupérez le dossier `2/` à trouver dans le dossier `algo2_src/tp/`.

Un module graphique, `sg`, figure dans le sous-dossier `sg/`. Ce module fort limité est basé sur X11 [6]. Vous pouvez exécuter et modifier à souhait l'exemple de programme de démonstration proposé qui utilise le module : `sg_test.c`.

Un deuxième module, `point`, figure dans le sous-dossier `point/`. Des représentations graphiques de qualité nécessitant des calculs ne peuvent être obtenues que lorsque ces calculs sont effectués sur les réels virgule flottante. Le module `point` définit le type `point` comme une structure à deux composantes de type `double`, propose quelques fonctions d'obtention de points particuliers à partir de deux autres et adapte, en les simplifiant, les procédures de traçage et de remplissage du module `sg`. Vous pouvez exécuter et modifier à souhait l'exemple de programme de démonstration proposé qui utilise le module `point` : `point_test.c`.

Un troisième module, `fractal`, figure dans le sous-dossier `fractal/`. Il est embryonnaire : il ne permet essentiellement que la production d'une figure fractale d'ordre donné, à savoir « les carrés emboités ». De même le source `main.c` qui contient la fonction principale : il ne fait essentiellement qu'appeler la fonction de production de la figure.

Exercice 1

Faites produire une deuxième figure fractale connue : spécification et déclaration de la fonction associée dans `fractal.h`, implantation dans `fractal.c`, appel initial dans `main.c`.

Suggestion de sources d'inspiration : les figures fractales montrées par l'exécutable `fractal` que vous trouverez sur UniversiTICE ; la page `https://fr.wikipedia.org/wiki/Liste_de_fractales_par_dimension_de_Hausdorff`.

Exercice 2

Faites produire d'autres figures fractales connues.

6. Si vous travaillez sur votre propre machine sous Ubuntu ou Xubuntu, X11, le *X Window System*, doit être installé. Si ce n'est pas déjà le cas, donnez du

```
sudo apt-get install libx11-dev
```

en ligne de commande.

Algorithmique 2 : structures de données linéaires
Fiche de TP n° 3

Programmation récursive et arithmétique des pointeurs

Objectifs : codage d'opérations récursives et utilisation de l'arithmétique des pointeurs.
Prérequis : arithmétique des pointeurs en C ; exercice 1.4 du support de cours et d'exercices.
Travail minimum : exercices 1 et 2.

Commencez par récupérer le dossier `3/` à trouver dans le dossier `algo2_src/tp/`. Ses sous-dossiers `str_rec/`, `quicksort/` et `balpar/` sont à usage exclusif des exercices 1, 2 et 3. Les trois exercices sont indépendants.

Dans les exercices 1 et 3, vous considèrerez les chaines de caractères comme des objets récursifs : étant donnée une chaine de caractères, soit `s` le pointeur de `char` qui la repère ; si la déréférence de ce pointeur, `*s` donc, a pour valeur le caractère nul `'\0'`, la chaine est vide ; sinon, la chaine est la composition d'un caractère (non nul), `*s`, et de la chaine de caractères repérée par le pointeur dont la valeur est l'adresse de type `char *` qui suit immédiatement `s`, autrement dit `s + 1`.

Exercice 1

Donnez vos propres versions des fonctions suivantes de la bibliothèque standard [7] :

```
#include <string.h>
char *strchr(const char *s, int c);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
```

Extraits du document ISO/IEC 9899:2023 (E) :

> *The* `strchr` *function.*
> *Description. The* `strchr` *function locates the first occurrence of* `c` *(converted to a* `char`*) in the string pointed to by* `s`*. The terminating null character is considered to be part of the string.*
> *Results. The* `strchr` *function returns a pointer to the located character, or a null pointer if the character does not occur in the string.*
>
> *The* `strpbrk` *function.*
> *Description. The* `strpbrk` *function locates the first occurrence in the string pointed to by* `s1` *of any character from the string pointed to by* `s2`*.*
> *Results. The* `strpbrk` *function returns a pointer to the character, or a null pointer if no character from* `s2` *occurs in* `s1`*.*
>
> *The* `strrchr` *function.*
> *Description. The* `strrchr` *function locates the last occurrence of* `c` *(converted to a* `char`*) in the string pointed to by* `s`*. The terminating null character is considered to be part of the string.*
> *Results. The* `strrchr` *function returns a pointer to the character, or a null pointer if* `c` *does not occur in the string.*

7. Un énoncé quasi identique figure sur le support de TP d'*Algorithmique 1*. Les fonctions devaient y être construites avec des boucles à l'aide de la logique de Hoare. Il s'agit ici de les définir récursivement.

The `strspn` *function.*

Description. The `strspn` *function computes the length of the maximum initial segment of the string pointed to by* `s1` *which consists entirely of characters from the string pointed to by* `s2`*.*

Results. The `strspn` *function returns the length of the segment.*

Contraintes :

— vos fonctions doivent être récursives terminales ou alors doivent faire appel à des fonctions auxiliaires qui le sont ;

— toute fonction auxiliaire doit être spécifiée ;

— afin de ne pas masquer les fonctions standards, les fonctions concurrentes doivent être nommées `str_chr`, `str_pbrk`, `str_rchr` et `str_spn` (déjà fait dans l'en-tête `"str_rec.h"`).

Le programme `main.c` propose dans sa version originelle quelques tests sur les quatre fonctions standard citées. L'accent circonflexe, l'astérisque et le point ont été choisis pour indiquer le résultat des recherches : le premier symbole marque la position de l'occurrence (recherche positive) pour les fonctions `strchr`, `strpbrk` et `strrchr` ; le second signifie une recherche négative pour ces mêmes fonctions ; le nombre de répétitions du troisième égale la longueur du préfixe pour la fonction `strspn`. Ces symboles sont suivis d'un rappel de l'objet qui guide la recherche.

Testez chacune des fonctions `str_`*suffixe* implantées. Pour ce faire : considérez chacune lignes de la fonction principale `main` prefixée par `TEST_` ; dupliquez-la (ctrl+d par exemple) ; sur la ligne copiée, remplacez l'occurrence de la fonction standard `str`*suffixe* par `str_`*suffixe*. Pour la première des quatre fonctions citées par exemple, cela doit donner :

```
TEST_PTR(strchr, s, 'd');
TEST_PTR(str_chr, s, 'd');
TEST_PTR(strchr, s, 'a');
TEST_PTR(str_chr, s, 'a');
TEST_PTR(strchr, s, '\0');
TEST_PTR(str_chr, s, '\0');
```

Si, à l'exécution, vous n'obtenez pas les mêmes résultats pour `str`*suffixe* et `str_`*suffixe*, c'est que votre implantation est à revoir. Dans le cas contraire, elle devrait être correcte ; vous pouvez dans ce cas supprimer les lignes originelles qui contiennent l'occurrence de `str`*suffixe*. Dès que toutes les fonctions semblent correctement implantées, vous pouvez supprimer l'inclusion de l'en-tête standard `<string.h>`.

Exercice 2

Implantez la version partiellement dérécursifiée du tri rapide qui figure dans le cours ; n'introduisez que des variables locales de type `char *`. Implantez sa fonction auxiliaire, l'algorithme de partition autour d'un pivot, de manière itérative et en conformité avec l'énoncé de l'exercice 1.4 du support de cours et d'exercices : même prototype, même spécification (déjà portés dans l'en-tête `"quicksort.h"`).

Le programme `main.c` se propose de tester visuellement votre réalisation sur un tableau de 25 entiers pseudo-aléatoires compris entre 0 et 99. Tapez ctrl+d ou alors q ou Q puis entrée pour quitter le programme, toute autre suite de caractères puis entrée pour enchainer sur un autre test.

Si la macroconstante `QSORT` est définie (c'est originellement le cas dans le fichier `makefile`), c'est la fonction standard `qsort` qui est appelée. Dès que votre version du tri rapide est prête à être testée, supprimez la définition de `QSORT` dans le fichier `makefile` puis construisez. Si votre version du tri rapide n'est pas prête à être testée mais que l'est celle de la partition autour d'un pivot, faites comme précédemment en remplaçant momentanément la valeur de la macroconstante `SORT` par `partition_pivot` dans la clause « sinon » de la directive `#if defined QSORT`.

Exercice 3

Une expression est dite *bien parenthésée* si elle est vide ou alors si elle est de la forme

$(e)e'$

où e et e' sont elles-même des expressions bien parenthésées[8].

Définissez la fonction `balpar` (« *balanced parentheses* ») déclarée dans l'en-tête `"balpar.h"`. Vous placerez cette définition dans le source `balpar.c` que vous créerez à cet effet. Vous développerez un programme de test, `main.c`. Le fichier `makefile` fourni est prévu pour fonctionner avec `"balpar.h"`, `balpar.c` et `main.c`.

Un exemple d'exécution attendu :

```
$ cat > test.txt
()()
(((())()))
(()
())[]
():-)
$ ./balpar < test.txt
()()
true
(((())()))
true
(()
^false
())[]
  ^false
():-)
  ^false
```

Attention : il ne s'agit en aucun cas ici d'établir le diagnostic de bon parenthésage en se basant sur un décompte des parenthèses ou sur l'utilisation explicite d'une pile.

Exercice 4

Allez plus loin que l'exercice précédent quant aux expressions supportées : les formes non vides peuvent faire apparaitre plusieurs paires de symboles associés, formées d'un symbole ouvrant et d'un symbole fermant. En mathématiques par exemple sont classiquement employées les paires « `(` » et « `)` », « `[` » et « `]` », « `{` » et « `}` ».

Vous pouvez développer des options, comme le choix des paires de symboles, ou permettre la lecture dans des fichiers. Un exemple d'exécutable qui met en œuvre ces possibilités figure sur UniversiTICE.

Pour ne pas perturber votre production à l'exercice 3, vous créerez un nouveau sous-dossier, `balpar_more/`, indépendant de `balpar/`, et travaillerez dedans. Modifiez le fichier `3/makefile` en conséquence.

8. Il est proposé dans cet exercice de ne s'intéresser qu'aux mots de Dyck sur l'alphabet des seules parenthèses ouvrante « `(` » et fermante « `)` ». La grammaire associée à cette restriction du langage de Dyck est $d \to \varepsilon \mid (d)d$. L'extension à d'autres paires d'ouvrants et fermants est proposée à l'exercice suivant.

Algorithmique 2 : structures de données linéaires

Fiche de TP n° 4

Listes dynamiques simplement chainées (1)

Objectifs : réalisation d'une implantation basique des listes dynamiques simplement chainées avec contrôleur.

Prérequis : allocation dynamique et chainage.

Travail minimum : exercices 1 et 2.

Commencez par récupérer le dossier `4/` à trouver dans le dossier `algo2_src/tp/`.

Placez-vous dans le sous-dossier `slower/`. Prenez connaissance du contenu du fichier en-tête `"slower.h"`. Pour comprendre l'utilisation qui peut être faite du module `slower`, vous pouvez regarder le contenu du source `slower_test.c`, produire l'exécutable associé, exécuter ce dernier en ligne de commande, modifier les valeurs des macroconstantes `LOWER_CARD`, `SLOWER_LENGTH` et `SLOWER_COMPAR_LENGTH` dans le fichier `makefile`, produire à nouveau l'exécutable, etc.

Par la suite, vous êtes autorisé à définir à la compilation chacune des trois macroconstantes sus-citées, à employer le type `slower`, par exemple pour déclarer des variables de ce type, et à faire appel aux quatre fonctions préfixées par `slower_` spécifiées et déclarées dans `"slower.h"`. Il vous est en revanche interdit de chercher à avoir accès au codage de tout objet de type `slower` : vous devez vous contenter des quatre fonctions ; ledit codage est à usage exclusif du source `slower.c`. Vous ne modifierez ni le fichier en-tête `"slower.h"`, ni le fichier `slower.c`.

Placez-vous dorénavant dans le sous-dossier `lslower/`. Prenez connaissance du contenu du source `main.c`. Produisez l'exécutable `lslower`. Exécutez ce dernier en ligne de commande. Conformément au contenu du source et à la valeur par défaut de macroconstante `SLOWER_LENGTH`, (seules) des lignes de mots de quatre lettres sont envoyées sur la sortie standard, une ligne de mots par tour de boucle **`do`**-**`while`** exécutée.

Prenez connaissance du contenu du fichier en-tête `"lslower.h"`, puis de la définition de la structure `lslower` donnée dans le fichier `lslower.c`, puis du reste du contenu de ce source.

Sans surprise aucune, vous devez dans un premier temps — exercice 1 — compléter le source `lslower.c` en définissant toutes les fonctions déclarées dans le fichier en-tête `"lslower.h"` et, conjointement, au fur et à mesure de l'avancement de ce travail, testez sous `valgrind` chacune de ces fonctions en les appelant à l'intérieur de la boucle **`do`**-**`while`** de la fonction `main` du source `main.c`. Un prolongement est proposé ensuite — exercice 2 —, lequel ne doit être abordé qu'à la condition expresse d'un développement parfait du module `lslower`.

Exercice 1

Complétez le source `lslower.c` en y définissant toutes les fonctions déclarées dans l'en-tête `"lslower.h"` tout en testant ces fonctions à partir du source `main.c`. Un plan de développement cohérent en six étapes est exposé après que des contraintes et des conseils aient été signifiés.

Contraintes :

— il vous est interdit de modifier les structures définies dans le fichier `lslower.c` ;

— il vous est imposé d'implanter les cinq fonctions `lslower_empty`, `lslower_is_empty`, `lslower_head_value`, `lslower_insert_head` et `lslower_move_head_head` de telle sorte qu'elles s'exécutent en temps constant (`lslower_empty` et `lslower_is_empty` sont déjà implantées) ;

— il vous est imposé d'implanter la fonction `lslower_move_head_head` de telle sorte qu'elle ne procède à aucune opération d'allocation ou de désallocation ;

— il vous est imposé de suivre le plan de développement.

Conseils :

— faites des dessins pour tout ce qui concerne la partie implantation du module `lslower` ;

— sous Geany, en plus des sources `lslower.c` et `main.c` à compléter, gardez toujours ouverts les en-têtes `"slower.h"` et `"lslower.h"` : vous bénéficierez ainsi de la coloration syntaxique et du rappel des prototypes lors de la saisie.

Plan de développement :

1) Préliminaires.

Au tout début du corps de la boucle `do-while` de la fonction `main` :

— déclarez la variable `s` de type `lslower *` ;

— initialisez-la en appelant `lslower_empty` ;

— testez dans la foulée que l'initialisation a réussi.

Pour la mise en œuvre de ce test, recourez à la macrofonction `ON_ERROR_GOTO` :

— déclarez l'étiquette `dispose:` [9] immédiatement avant l'accolade fermante du corps de la boucle `do-while` ;

— immédiatement après l'affectation à `s`, appelez la macrofonction avec le test `s == nullptr` comme premier argument, la chaine mentionnant la cause de l'erreur, `"Heap overflow"`, comme deuxième argument et l'identificateur de l'étiquette comme troisième et dernier argument ;

MÀJ 15-10 `NULL` → `nullptr` (Ilyas)

— mettez en forme avec Uncrustify.

À la fin de l'exécution — et sauf à ce que l'espace mémoire disponible pour le tas ne soit pathologiquement réduit —, `valgrind` doit râler en indiquant dans sa rubrique « `HEAP SUMMARY` » qu'un nombre de désallocations égal au nombre de tours de boucle effectués n'ont pas été faites (les contrôleurs des listes n'ont effectivement pas été désalloués). Ce problème ne sera résolu qu'après une implantation correcte et une utilisation cohérente de la fonction `lslower_dispose`.

2) `lslower_dispose`.

Implantez la fonction `lslower_dispose` dans `lslower.c`.

Dans `main.c` :

— faites suivre la déclaration de l'étiquette introduite dans le 1) d'un appel à la fonction avec `&s` comme paramètre ;

— mettez en forme avec Uncrustify.

À la fin de l'exécution, `valgrind` ne doit plus râler : les nombres d'allocations et de désallocations doivent être égaux. Dans le cas contraire, allez en 2).

3) `lslower_fput` et `lslower_insert_head`.

Implantez les deux fonctions dans `lslower.c`.

Testez-les dans `main.c` :

— immédiatement avant l'accolade fermante du corps de la boucle `for`, insérez l'instruction `ON_ERROR_GOTO(lslower_insert_head(s, &x) != 0, "Heap overflow", dispose);` ;

— immédiatement après l'instruction `fputc('\n', stdout);` qui suit la boucle `for`, insérez l'instruction `lslower_fput(s, stdout);` [10] ;

— produisez l'exécutable, lancez en ligne de commande ;

— constatez. Si l'ordre des mots qui figurent sur chaque ligne produite par `lslower_fput` est à rebours de celui des mots qui figurent sur la ligne qui la précède immédiatement, ça semble gagné. Sinon allez en 3) : révisez votre implantation.

À la fin de l'exécution, `valgrind` ne doit toujours pas râler. Il doit en revanche indiquer des nombres d'allocations et de désallocations toujours égaux mais bien supérieurs à ceux de l'étape

9. Le choix de l'identificateur de l'étiquette devient clair à l'étape 2) du plan de développement.

10. Toutes les sorties effectuées par la fonction principale `main` sont à destination des sorties standard et erreur. Il est proposé ici de supposer qu'aucune erreur ne peut survenir lors de ces sorties, et donc de ne pas avoir à les tester.

précédente : $(1+N)\times n+2$, où N est la valeur de la macroconstante `NMEMB` déclarée dans `main.c` [11] et n le nombre de tours de boucles. Dans le cas contraire, allez en 2).

4) `lslower_head_value`.

Commencez par faire un peu de ménage dans `main.c` en supprimant les quatre lignes qui figurent entre les instructions `slower_rand(&x);` et `lslower_insert_head(s, &x);` ainsi que l'instruction `fputc('\n', stdout);` qui suit la boucle **`for`** (autrement dit les instructions qui servent au seul affichage des valeurs produites par `slower_rand`).

Implantez la fonction `lslower_head_value` dans `lslower.c`.

Testez-la dans `main.c` :

— insérez immédiatement après l'instruction `lslower_fput(s, stdout);` d'affichage de la liste en entier le bloc

```
{
  slower x;
  ON_ERROR_GOTO(lslower_head_value(s, &x) != 0, "Internal error", dispose);
  slower_fput(&x, stdout);
  fputc('\n', stdout);
}
```

— produisez l'exécutable, lancez en ligne de commande ;

— constatez. Si le premier mot de la liste se retrouve répété en dessous de lui-même, ça semble gagné. Sinon allez en 4).

À la fin de l'exécution, le bilan `valgrind` doit être de la même teneur qu'à l'étape précédente.

5) `lslower_move_head_head`.

Implantez la fonction dans `lslower.c`.

Dans `main.c`, testez en introduisant une deuxième liste :

— doublez l'instruction de déclaration et d'initialisation de la variable `s`. Dans la copie, remplacer `s` par `s2` ;

— faites de même pour l'instruction de libération des ressources liées à la variable `s` ;

— doublez le test d'erreur pour dépassement de capacité qui suit l'initialisation de la variable `s` en l'adaptant pour la variable `s2` ;

— remplacer le bloc introduit à l'étape précédente par les quatre lignes

```
while (lslower_move_head_head(s, s2) == 0) {
  lslower_fput(s, stdout);
  lslower_fput(s2, stdout);
}
```

— produisez l'exécutable, lancez en ligne de commande ;

— constatez. Si les mots de la première liste disparaissent au fur et à mesure dans la deuxième mais à rebours, ça semble gagné. Sinon goto 5).

Même chose que précédemment quant au bilan final établi par `valgrind`.

6) Apothéose temporaire.

Testez en donnant les valeurs 1 puis 0 à la macroconstante `NMEMB`.

Même chose que précédemment quant au bilan final établi par `valgrind`, sauf en ce qui concerne le qualificatif « bien supérieurs ».

Si vous réussissez à prolonger en partie ou en totalité avec l'exercice 2, fixez `NMEMB` à 16 et supprimez la boucle introduite au 5).

11. Jusques y compris l'avant dernière étape, il est supposé que la valeur de la macroconstante `NMEMB` est entière et supérieure ou égale à 2. Pour remplir au maximum une ligne d'un terminal selon de standard des 80 caractères en largeur, la valeur de `NMEMB` doit être fixée à 16 lorsque `SLOWER_LENGTH` vaut 4.

Exercice 2

Sous Geany, supprimez l'onglet correspondant au source `lslower.c` : vous n'en aurez plus besoin.

Prenez connaissance du contenu du fichier en-tête `"lslower_ext.h"`. À l'instar de l'exercice précédent, implantez dans le source `lslower_ext.c` et testez dans la fonction principale `main` du source `main.c`.

1) Définissez la fonction `lslower_move_all_head`.

Contraintes :

— aucune cellule ne doit être allouée ou désallouée ;

— en cas de succès, le temps d'exécution de la fonction doit être linéaire en la longueur de la liste associée à son premier argument.

Testez la fonction dans `main.c` en réutilisant le matériel du test du 5) de l'exercice 1 : (ne) basculez (plus que) quelques-uns des mots de `s` vers `s2` avec `lslower_move_head_head` (`NMEMB / 3` mots par exemple), puis concaténez à `s2` ce qui reste de `s` avec `lslower_move_all_head`.

Si la fonction donne entière satisfaction, ajoutez un « s » au « module » qui apparait dans la première instruction de la fonction `main`.

2) Définissez la fonction `lslower_partition_pivot`.

Les contraintes sont les mêmes que précédemment, sauf que le succès est ici garanti.

Élaborez un test probant (en plus de `s`, utilisez trois autres variables de type `lslower *` ; pour obtenir plus assurément des listes des égaux non réduites aux seuls pivots, donnez à la macro-constante `SLOWER_COMPAR_LENGTH` une valeur strictement inférieure à `SLOWER_LENGTH`, par exemple `SLOWER_COMPAR_LENGTH / 2`).

3) Définissez la fonction `lslower_quicksort`. Élaborez un test probant (retour à la seule variable `s`).