Programmation impérative (avancé)
Projet

Licence informatique 2ème année

Université de La Rochelle

<christophe.demko@univ-lr.fr>

2022-2023_2

Les lignes commençant par $ représente une commande *unix*. Le $ ne fait pas partie de la commande proprement dite mais symbolise l'invite de commande.

Ce projet a pour vocation à devenir une librairie efficace de gestion des listes d'inversion qui sera intégré au projet **GALACTIC** mené par le laboratoire *L3i*.

Table des matières

Liste des exercices

1 Consignes

1.1 Groupes

Le projet se fait par groupe de maximum 6 étudiants et est à rendre par un dépôt *moodle* le vendredi 19 mai 2023 à 23h00. Un retard raisonnable est toléré mais il sera pénalisé.

1.2 Langue utilisée

La langue utilisée dans le code et la documentation devra être exclusivement l'anglais. Si vous avez des difficultés dans la langue de Shakespeare, vous pourrez utiliser les traducteurs automatiques :

- https://www.deepl.com/translator
- https://translate.google.fr/
- https://chat.openai.com/

1.3 Nommage

1.3.1 Fichiers

- les noms de fichiers du langage C seront tous en minuscules et en anglais. S'ils sont composés de plusieurs mots, ils devront être séparés par un tiret (−) (notation *kebab case* https://en.wikipedia.org/wiki/Letter_case#Kebab_case);
- les fichiers contenant le code devront avoir l'extension `.c` ;
- les fichiers d'en-têtes (exportables) devront avoir l'extension `.h` ;
- les fichiers destinés à être inclus dans votre code mais non exportables devront avoir l'extension `.inc`

1.3.2 Types

Les noms de types devront faire commencer chaque mot qui les compose par une majuscule. Il n'y a pas de sous-tirets (notation *Pascal case* https://en.wikipedia.org/wiki/Letter_case#Camel_case). Les structures devront commencer par un sous-tiret (_) puis suivre la notation *Pascal case* pour ne pas les confondre avec les noms de types.

1.3.3 Macros

Les macros (avec ou sans arguments) s'écrivent tout en majuscule en séparant les mots par des sous-tirets (_) (notation *screaming snake case* https://en.wikipedia.org/wiki/Letter_case#Snake_case).

1.3.4 Variables et fonctions

Les variables et les fonctions s'écrivent toutes en minuscules en séparant les mots par des sous-tirets (notation *snake case* https://en.wikipedia.org/wiki/Letter_case#Snake_case).

1.4 Style

1.4.1 Marque d'inclusion unique

Chaque fichier d'en-tête devra posséder une marque permettant d'éviter les conséquences d'un fichier inclus plusieurs fois. Voir https://google.github.io/styleguide/cppguide.html#The__define_Guard

1.4.2 Ordre des inclusions

L'inclusion des fichiers d'en-tête devra respecter la logique suivante :

1. Inclusion du fichier directement lié au fichier `.c` qui l'inclut suivi d'une ligne vide ;
2. inclusion des fichiers d'en-tête du C standard suivis d'une ligne vide ;
3. inclusion des fichiers d'en-tête provenant d'autres librairies suivis d'une ligne vide ;
4. inclusion des fichiers d'en-tête du projet suivi d'une ligne vide ;
5. inclusion des fichiers d'inclusion (extension `.inc`)

1.4.3 Indentation

Le style d'indentation devra être celui préconisé par Google https://google.github.io/styleguide/cppguide.html#Formatting. L'utilitaire `clang-format` (https://clang.llvm.org/docs/ClangFormat.html) supporte le style `Google`.

Vous pourrez utiliser l'utilitaire `cclint` pour vérifier votre code.

1.5 Structuration du code

Les champs des structures seront protégés à la manière de la librairie `fraction` vue en travaux pratiques. Un soin sera tout particulièrement apporté à la structuration du code notamment en ce qui concerne les structures de données utilisées.

1.6 Documentation

La documentation sera générée avec l'outil `sphinx` et les fonctions seront documentées avec la norme de `doxygen` pour le langage C.

1.7 Tests

Des tests unitaires devront être implémentés, ils testeront chaque fonction et s'efforceront de vérifier que la mémoire est bien libérée au moyen de l'utilitaire `valgrind`.

Vous pourrez vous inspirer du projet https://github.com/chdemko/c-arithmetic.

D'une manière générale, toutes les options possibles décrites dans ce projet (https://github.com/chdemko/c-arithmetic) devront être implémentées.

2 Sujet

Le but du projet est de produire :

- une librairie permettant d'ajouter des fonctionnalités aux listes d'inversion ;
- une librairie *python* appelant, pour les calculs, la librairie du langage C.

Le projet devra fournir une documentation produite avec

```
$ make docs
```

Il pourra être installé avec

```
$ make install
```

La couverture de son code pourra être calculée avec

```
$ make html-coverage
```

Pour la librairie *python* la structure du code devra être semblable au projet https://github.com/chdemko/py-arithmetic.

Il est tout à fait possible de recourir à des logiciels de génération de code du style de https://docs.github.com/fr/copilot.

Exercice 1 *(Optimisation du type `InversionList`)*
Lorsque la capacité de l'univers est de petite taille (≤ 256 ou ≤ 65536) il peut être utile de stocker les entiers composant les couples de la liste d'inversion par des entiers sur 1 ou 2 octets.
Modifier la structure inversion permettant de choisir la taille utilisée pour chaque entier composant les couples en fonction du champs `capacity` :

- l'accès aux couples (dans le fichier `inversion-list.c`) se fera avec la construction suivante :

  Si la capacité est ≤ 256 :

  ```
  InversionList *set = inversion_list_create(100, ...);
  int index;
  ...
  set->couples.uint8[index] = ...
  ```

  Si la capacité est ≤ 65536 :

  ```
  InversionList *set = inversion_list_create(1000, ...);
  ```

```
int index;
...
set->couples.uint16[index] = ...
```

Sinon :

```
InversionList *set = inversion_list_create(100000, ...);
int index;
...
set->couples.uint32[index] = ...
```

- les champs `uint8`, `uint16`, `uint32` **partagent** la même mémoire et seront respectivement du type `uint8_t`, `uint16_t`, `uint32_t` de l'entête standard `<stdint.h>` ;
- la création de la structure ne comporte toujours qu'une seule allocation ;
- les déclarations de fonction de l'entête `inversion-list.h` ne seront pas modifiées.

Modifier le code des autres fonctions en conséquence.

Exercice 2 *(Fonctions de comparaison)*
Écrire les fonctions suivantes :

```
extern bool inversion_list_equal(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_not_equal(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_less(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_less_equal(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_greater(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_greater_equal(
  const InversionList *set1,
  const InversionList *set2
);
extern bool inversion_list_disjoint(
  const InversionList *set1,
  const InversionList *set2
);
```

permettant d'affirmer ou d'infirmer respectivement que :

- `set1` est égal à `set2` ;
- `set1` est différent de `set2` ;
- `set1` est plus petit que `set2` (au sens de l'inclusion ensembliste) ;
- `set1` est plus petit ou égal à `set2` (au sens de l'inclusion ensembliste) ;
- `set1` est plus grand que `set2` (au sens de l'inclusion ensembliste) ;
- `set1` est plus grand ou égal à `set2` (au sens de l'inclusion ensembliste) ;
- `set1` est disjoint de `set2`.

Vous vous efforcerez d'optimiser ces fonctions pour les rendre les plus efficaces possibles.

Exercice 3 *(Fonctions de calcul à 2 arguments de complexité linéaire)*
Écrire les fonctions suivantes ayant une complexité linéaire (Complexité des algorithmes) :

```
extern InversionList *inversion_list_union(
  const InversionList *set1,
  const InversionList *set2
);
extern InversionList *inversion_list_intersection(
  const InversionList *set1,
  const InversionList *set2
);
extern InversionList *inversion_list_difference(
  const InversionList *set1,
  const InversionList *set2
);
extern InversionList *inversion_list_symmetric_difference(
  const InversionList *set1,
  const InversionList *set2
);
```

permettant, en utilisant des algorithmes linéaires, de calculer respectivement :

- l'union de `set1` et `set2` (au sens ensembliste) ;
- l'intersection de `set1` et `set2` (au sens ensembliste) ;
- la différence de `set1` et `set2` (au sens ensembliste) ;
- la différence symétrique de `set1` et `set2` (au sens ensembliste).

Exercice 4 *(Fonctions de calcul à nombre d'arguments variables)*
Modifier les fonction suivantes pourqu'elles puissent prendre un nombre variable d'arguments (se terminant par `NULL`) :

```
extern InversionList *inversion_list_union(
  const InversionList *set,
  ...
);
extern InversionList *inversion_list_intersection(
  const InversionList *set,
```

```
  ...
);
extern InversionList *inversion_list_difference(
  const InversionList *set,
  ...
);
```

permettant, en utilisant des algorithmes linéaires, de calculer respectivement :

- l'union de `set` et des ensembles supplémentaires (au sens ensembliste) ;
- l'intersection de `set` et des ensembles supplémentaires (au sens ensembliste) ;
- la différence de `set` et des ensembles supplémentaires (au sens ensembliste).

Exercice 5 *(Fonctions de calcul dichotomiques)*
Les fonctions écrites calculent leur résultats en parcourant l'ensemble des couples à leur disposition. Il peut être intéressant d'utiliser la recherche dichotomique afin d'améliorer la complexité moyenne (au détriment de la complexité du pire des cas). Modifier les fonctions de calcul dans ce sens. Donner une explication des complexités algorithmiques obtenues.

Exercice 6 *(Itérateurs sur les entiers)*
Proposer des fonctions et un type permettant de manipuler un itérateur sur les entiers d'une liste d'inversion.

```
typedef struct _InversionListIterator InversionListIterator;
extern InversionListIterator *inversion_list_iterator_create(
  const InversionList *set
);
extern void inversion_list_iterator_destroy(
  InversionListIterator *iterator
);
extern InversionListIterator *inversion_list_iterator_next(
  InversionListIterator *iterator
);
extern InversionListIterator *inversion_list_iterator_rewind(
  InversionListIterator *iterator
);
extern bool inversion_list_iterator_valid(
  const InversionListIterator *iterator
);
extern unsigned int inversion_list_iterator_get(
  const InversionListIterator *iterator
);
```

Exercice 7 *(Itérateurs sur les couples)*
Proposer des fonctions et un type permettant de manipuler un itérateur sur les couples d'une liste d'inversion.

```
typedef struct _InversionListCoupleIterator InversionListCoupleIterator;
extern InversionListCoupleIterator *inversion_list_couple_iterator_create(
```

```
  const InversionList *set
);
extern void inversion_list_couple_iterator_destroy(
  InversionListCoupleIterator *iterator
);
extern InversionListIterator *inversion_list_couple_iterator_next(
  InversionListCoupleIterator *iterator
);
extern InversionListCoupleIterator *inversion_list_couple_iterator_rewind(
  InversionListCoupleIterator *iterator
);
extern bool inversion_list_iterator_couple_valid(
  const InversionListCoupleIterator *iterator
);
extern unsigned int inversion_list_couple_iterator_get_inf(
  const InversionListCoupleIterator *iterator
);
extern unsigned int inversion_list_couple_iterator_get_sup(
  const InversionListCoupleIterator *iterator
);
```

Exercice 8 *(Librairie python)*

En prenant exemple sur le projet https://github.com/chdemko/py-arithmetic, proposer une extension python, utilisant la librairie écrite en C, permettant de gérer les ensembles d'entiers à univers fini en utilisant les conventions suivantes :

- la librairie devra se situer dans un *package* `finite_set` et contiendra une class `IntegerSet` ;
- la classe `IntegerSet` devra implémenter la classe `AbstractSet[int]` :

```
class IntegerSet(AbstractSet[int]):
    def __init__(
        self,
        intervals: Optional[Iterable[Tuple[int, int]]] = None,
    ) -> None: ...
    @classmethod
    def from_iterable(
        cls,
        iterable: Optional[Iterable[int]] = None,
    ) -> IntegerSet: ...
    def __repr__(self) -> str: ...
    def __hash__(self) -> int: ...
    def __contains__(self, item: object) -> bool: ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
    def intervals(self) -> Iterator[Tuple[int, int]]: ...
    def ranges(self) -> Iterator[range]: ...
    def __eq__(self, other: object) -> bool: ...
    def __ne__(self, other: object) -> bool: ...
```

```
    def __lt__(self, other: "IntegerSet") -> bool: ...
    def __le__(self, other: "IntegerSet") -> bool: ...
    def __gt__(self, other: "IntegerSet") -> bool: ...
    def __ge__(self, other: "IntegerSet") -> bool: ...
    def isdisjoint(self, other: Iterable[int]) -> bool: ...
    def __and__(self, other: "IntegerSet") -> "IntegerSet": ...
    def intersection(self, *others: Iterator[int]) -> "IntegerSet": ...
    def __or__(self, other: "IntegerSet") -> "IntegerSet": ...
    def union(self, *others: Iterator[int]) -> "IntegerSet": ...
    def __sub__(self, other: "IntegerSet") -> "IntegerSet": ...
    def difference(self, *others: Iterator[int]) -> "IntegerSet": ...
    def __xor__(self, other: "IntegerSet") -> "IntegerSet": ...
    def symmetric_difference(self, other: Iterator[int]) -> "IntegerSet":
    ↪   ...
    def intervals(self) -> Iterator[Tuple[int, int]]: ...
    def ranges(self) -> Iterator[range]: ...
```

La librairie *python* sera documentée en utilisant l'outil *sphinx* et le thème *sphinx_rtd_theme*.

Exercice 9 *(Critique de l'approche)*
Quel est l'inconvenient majeur de l'approche proposée utilisant les listes d'inversion pour gérer les ensembles d'entiers ?
Vous décrirez votre réponse dans la documentation générée à l'aide de la commande

```
$ make docs
```

Historique des modifications

2022-2023_1 *Dimanche 12 mars 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Version initiale

2022-2023_2 *Mercredi 18 avril 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Correction dans l'exercice 7 (itérateurs sur les couples)
- Précision dans l'exercice 8 (librairie python)

