TP noté de programmation impérative

ÉNSIIE, semestre 1

mercredi 15 décembre 2021

Durée : 2h30.
Les exercices 2 et 3 sont indépendants, et peuvent donc être traités dans l'ordre voulu. Un code qui ne compile pas entraîne une note nulle. La présence de *warnings* à la compilation entraîne une perte de points.

Vous trouverez à l'emplacement `/pub/IPI/TP_note/VotreNom_VotrePrenom/` des fichiers `stack_`i`.h` et `stack_`i`.c` contenant une interface et une implémentation des piles. Il s'agit d'une version personnalisée du module tel que vu en cours. Vous devrez utiliser ces fichiers pour réaliser votre TP, mais vous ne devrez en aucun cas les modifier. Le reste de votre code devra être écrit en accord avec ces fichiers.

Au milieu de la session (10h45), vous déposerez une archive `.tar.gz` contenant l'état courant de votre travail (même s'il ne compile pas encore) dans le dépot `ipi_tp_note_intermediaire_2021` sur http://exam.ensiie.fr/ . Le travail dans ce dépôt ne sera pas noté, mais son absence entraînera une note nulle.

À la fin de la session vous déposerez une archive `.tar.gz` contenant votre travail dans le dépôt `ipi_tp_note_final_2021` sur http://exam.ensiie.fr/ . Seul ce dépôt sera noté. Vous veillerez à ce que votre archive contienne bien les fichiers attendus, aucune erreur ne sera tolérée.

Tous les documents, cours, TD et TP sont autorisés. Les dépôts seront comparés deux à deux. En cas de similitudes entre les codes (y compris, mais pas seulement, si vous vous êtes contenté de recopier un code en changeant le nom des variables et/ou les commentaires), les deux auteurs se verront attribuer la note 0 à l'ensemble du TP, y compris si la similitude ne porte que sur une question.

Exercice 1 : Ensembles

Un ensemble est une structure de données abstraite qui contient une collection d'éléments telle qu'il n'y a pas d'éléments en double, et telle que l'ordre dans lequel ont été insérés les éléments n'est pas important.

Dans ce sujet, on considérera des ensembles contenant des `char`. L'interface du module pour les ensembles, qui sera écrite dans un fichier `set.h` (attention à bien respecter ce nom pour que le fichier `stack_`i`.h` compile correctement), comportera donc :

- la définition d'un type concret pour les valeurs contenues dans les ensembles, à savoir des `char` ;
- la déclaration d'un type **abstrait** pour les ensembles, que l'on appellera **obligatoirement** `set` pour que le fichier `stack_`i`.h` compile correctement ;
- la déclaration d'une fonction `empty_set` qui ne prend pas de paramètre et qui retourne un ensemble (à savoir l'ensemble vide) ;
- la déclaration d'une procédure `add_to_set` qui prend en paramètre un élément et un ensemble par référence, et qui ajoute par effet l'élément dans l'ensemble ;

— la déclaration d'une procédure `remove_from_set` qui prend en paramètre un élément et un ensemble par référence, et qui retire par effet l'élément dans l'ensemble ; si l'élément n'était pas présent, il ne se passe rien ;

— la déclaration d'une fonction `set_union` qui prend en paramètre deux ensembles et qui retourne leur union ; cette fonction sera autorisée à modifier les ensembles passés en paramètre ;

— la déclaration d'une fonction `set_difference` qui prend en paramètre deux ensembles et qui retourne leur différence ; cette fonction sera autorisée à modifier les ensembles passés en paramètre ; on rappelle que si $A = \{a; b; c\}$ et $B = \{b; d\}$, alors la différence $A \setminus B$ est égale à $\{a; c\}$: on retire du premier ensemble les éléments du deuxième s'ils étaient présents ;

— la déclaration d'une procédure `print_set` qui prend en paramètre un ensemble et qui l'affiche sur la sortie standard ; cet affichage se fera de la forme `{ a; b; d; f }` ;

— la déclaration d'une fonction `set_equal` qui prend en paramètre deux ensembles et qui teste si ces ensembles sont égaux (égalité ensembliste : ils ont les mêmes éléments, quelque soit l'ordre dans lequel ils ont été insérés).

1. Écrire cette interface dans un fichier `set.h`. Attention, ici **on n'écrira pas encore l'implémentation concrète !**

Les deux sections suivantes sont indépendantes. Néanmoins, il faudra avoir réalisé les deux pour avoir un programme qui fonctionne.

Exercice 2 : Manipulation d'ensembles

Le but de ce TP est d'écrire une fonction qui permet de créer des ensembles en lisant une chaîne de caractères dans un certain format. Pour cela, on va utiliser des piles dont les éléments **sont des ensembles**. Autrement dit, ces piles contiendront des ensembles, et ces ensembles contiendront chacun des `char`.

On part d'une pile vide. Étant donnée une chaîne de caractère, on va la parcourir de gauche à droite et agir ainsi successivement sur chaque caractère :

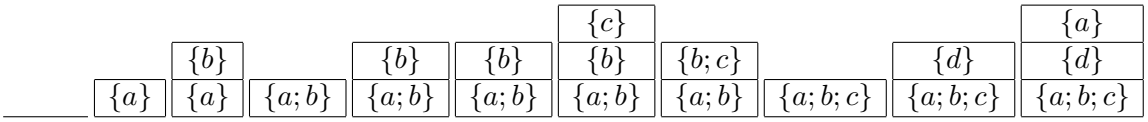
— si le caractère courant est compris entre `'a'` et `'z'`, on va rajouter au sommet de la pile un ensemble qui contient ce caractère et uniquement ce caractère (un ensemble singleton donc) ;

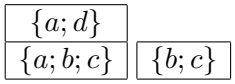
— si le caractère courant est `'+'`, on va dépiler deux ensembles, puis empiler leur union ;

— si le caractère courant est `'-'`, on va dépiler deux ensembles A puis B, puis empiler leur différence $B \setminus A$ (attention à l'ordre) ;

— si c'est un autre caractère, on ne fait rien.

Ainsi, sur la chaîne de caractère `"ab+b c++da+-"` on aura un pile qui évolue de la façon suivante :





.

On considérera donc trois modules :

— un module `main.c` qui implémentera cette fonction ;

— le module `stack_`i`.c` fourni qui implémente des piles dont les éléments sont des ensembles ;

— le module `set.c` dont l'interface est décrite dans la section précédente.

2. Écrire un fichier `Makefile` pour le projet, que l'on complétera au fur et à mesure de l'avancement. Au final, on veut produire un exécutable appelé `set_compute` qui réunira ces trois modules.

3. Dans un fichier `main.c`, écrire une fonction `parse_string` qui prend en paramètre une chaîne de caractère et qui retourne l'ensemble situé au sommet de la pile après avoir parcouru la chaîne comme décrit ci-dessus.

4. Écrire une fonction `main` qui lit deux lignes sur l'entrée standard, qui applique la fonction `parse_string` sur chacune pour obtenir deux ensembles, puis qui affiche ces ensembles en indiquant s'ils sont égaux ou non.

   Exemple de comportement attendu :

   ```
   $ ./set_compute
   ab+b c++da+-
   cbbb++c++
   Les ensembles { b; c } et { c; b } sont égaux.
   ```

   (En fonction de votre implémentation, il se peut que l'ordre dans lequel les éléments seront affichés soit différent de celui de l'exemple, mais cela ne posera pas de problème.)

Exercice 3 : Listes sans doublons

Une façon relativement simple, mais pas très efficace, d'implémenter les ensembles est d'utiliser des listes chaînées dans lesquelles on s'assure qu'il n'y a pas de doublon : quand on insère un élément dans la liste, on vérifiera qu'il n'est pas déjà présent dans la liste.

Dans un fichier `set.c`

5. Définir le type concret vers lequel pointent les ensembles.

   Indication : il s'agit d'un maillon de la liste chaînée.

6. Définir la fonction `empty_set` qui retourne un ensemble vide.

   Indication : il s'agit de la liste vide.

7. Définir la procédure `add_to_set`.

   Indication : on fait un parcours de la liste pour vérifier si l'élément est déjà présent ou non ; s'il l'est, on ne fait rien ; sinon on ajoute un nouveau maillon contenant l'élément, au choix en début ou en fin de liste.

8. Définir la procédure `print_set`. On fera attention à respecter le format demandé (exemples : `{ }`, `{ a }`, `{ a; f; c }`).

9. Définir la procédure `remove_from_set`.

   Indication : c'est la même procédure que pour retirer **un** élément d'une liste. En effet, on peut supposer que si l'élément est présent, alors il ne l'est qu'une seule fois.

10. Définir une première version de la fonction `set_union`. Pour cela, il suffit de parcourir la première liste en ajoutant chacun de ses éléments à la seconde à l'aide de la procédure `add_to_set`. On retourne alors la seconde liste.

11. Définir la fonction `set_difference`.

    Indication : on parcours la seconde liste en retirant chacun de ses éléments de la première à l'aide de la procédure `remove_from_set`. On retourne alors la première liste.

12. Définir une fonction `set_include` qui prend en paramètre deux ensembles et qui teste si le premier est inclus dans le second.

    Indication : on parcourt la première liste pour vérifier que chacun de ses éléments est dans la seconde.

13. Définir la fonction `set_equal`.

    Indication : deux ensembles sont égaux si chacun est inclus dans l'autre.

14. Définir une fonction `add_bucket` qui prend en paramètre un maillon de liste par référence et une liste, et qui ajoute ce maillon dans la liste si sa valeur n'est pas déjà présente dans la liste. On ne fera donc pas de nouvelles allocations dynamiques (y compris indirectement).

    On fera attention à ne bien inclure que ce maillon et pas les éventuels maillons auquel il serait attaché.

    Si la valeur associée au maillon est déjà présente dans la liste, on libérera la mémoire associée au maillon.

15. Utiliser cette fonction pour redéfinir la fonction `set_union` de façon à ce qu'elle ne fasse pas d'allocation mémoire.

    Indication : on va ajouter successivement tous les maillons de la première chaîne dans la seconde.

    On mettra la première version écrite question 10 en commentaire pour qu'elle soit prise en compte dans la correction.