TP noté de programmation impérative

ÉNSIIE, semestre 1

mercredi 20 décembre 2023

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 (hormis ceux dûs à l'obfuscation du fichier `stack_`i`.c`) entraîne une perte de points.

Vous trouverez à l'emplacement `/pub/FISE_PRIM11/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. Vous trouverez également un fichier `dns.c` qu'il vous faudra compléter à la partie 3. Copiez ces trois fichiers dans le répertoire dans lequel vous allez travailler avant de commencer.

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_2023` 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_2023` 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. Aucun échange, y compris électronique, n'est autorisé durant l'examen.

Arbres binaires de recherche

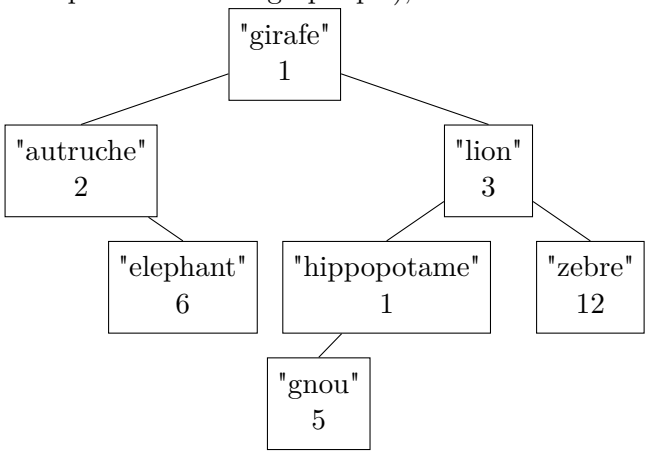
On rappelle qu'un dictionnaire (aussi appelé tableau associatif ou table d'association, en anglais *map*) est une structure de données abstraite qui permet de modéliser des fonctions (au sens mathématique) partielles discrètes : on associe à certains éléments d'un ensemble K (les clefs) des valeurs d'un ensemble V.

Nous avons vu en TP qu'une façon simple d'implémenter un dictionnaire est d'utiliser une liste d'associations. L'inconvénient est que la recherche d'un élément se fait dans le pire des cas en temps linéaire par rapport au nombre d'associations.

Pour limiter cela, on ne va donc pas utiliser une structure linéaire comme les listes chaînées, mais une structure arborescente : les arbres binaires de recherche. Un arbre binaire de recherche est un arbre dont chaque nœud contient une clef et une valeur associée, et a au plus deux fils, un à gauche et un à droite ; un arbre binaire de recherche doit respecter l'invariant suivant : la clef à sa racine est plus grande que toutes les clefs de son sous-arbre gauche s'il existe, elle est plus petite que toutes les clefs de son sous-arbre droit s'il existe, et

ses deux sous-arbres respectent également cet invariant. Ainsi, quand on cherchera une clef dans l'arbre, on pourra se contenter de chercher dans le sous-arbre gauche (resp. le sous-arbre droit) si la clef recherchée est strictement plus petite (resp. strictement plus grande) que celle à la racine.

Exemple d'arbre binaire de recherche dont les clefs sont des chaînes de caractères (ordonnées par ordre lexicographique), et dont les valeurs sont des entiers :



Au cours de ce TP, **les clefs seront des chaînes de caractères, et les valeurs des entiers**.

Exercice 1 : Interface

Dans un fichier `map.h`, écrire l'interface des dictionnaires. On y déclarera donc (**sans écrire d'implémentation ici**) :

1. des types concrets entiers pour les clefs (chaînes de caractères) et les valeurs (entiers) ;
2. un type abstrait `map` pour les dictionnaires ;
3. une fonction `map_empty` qui retourne un dictionnaire vide ;
4. une fonction `map_find` qui prend en paramètre une clef et un dictionnaire, et qui retourne la valeur associée à la clef dans le dictionnaire ; si aucune valeur n'est associée à la clef, on retournera `-1` ;
5. une procédure `map_add` qui prend en paramètre une clef, une valeur, et un dictionnaire par référence, et qui ajoute par effet l'association entre la clef et la valeur dans le dictionnaire ; si la clef est déjà associée à une valeur, on remplacera l'association ;
6. une procédure `map_print` qui prend en paramètre un dictionnaire et qui affiche les associations contenues dedans, une par ligne, de la forme :
   "*clef*" -> *valeur*
   On affichera dans l'ordre croissant des clefs.
7. une procédure `map_remove` qui prend en paramètre une clef et un dictionnaire par référence, et qui retire par effet du dictionnaire l'association avec la clef si elle existe ; sans effet sinon ;

Pour ce TP, on considérera trois modules :

— le module `stack_`*i*`.c` fourni ;

— un module `map.c` qui implémente les arbres binaires de recherche (exercice 2) ;
— un module `dns.c` à compléter, qui utilise les dictionnaires pour représenter des serveurs de noms de domaine dans une version simplifiée de la résolution de noms de domaine (exercice 3).

8. 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 `dns_resolve` qui réunira ces trois modules.

Exercice 2 : Implémentation

Dans un fichier `map.c`, écrire l'implémentation des dictionnaires en utilisant des arbres binaires de recherche.

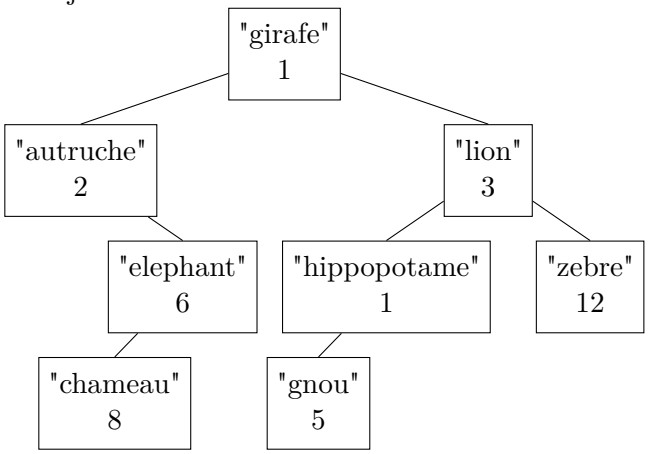
9. Définir un type `tree` qui sera un pointeur vers un enregistrement `struct node`, ce dernier étant composé de quatre champs : une clef `k`, une valeur `v`, un fils gauche `left` de type `tree` et un fils droit `right` de type `tree`.
10. Écrire une fonction `make_node` qui prend en paramètre une clef et une valeur, et qui retourne un arbre constitué d'un nœud nouvellement alloué contenant la clef et la valeur et n'ayant pas de fils. (Pour cela, on remplira les champs `left` et `right` avec la valeur `NULL`.)
11. Pour `map_empty`, il suffira de retourner `NULL` qui représente l'arbre vide.
12. Pour `map_find`, on utilisera l'algorithme suivant : si l'arbre est vide, on retourne `-1`, sinon si la clef de la racine est celle recherchée, on retourne la valeur de la racine, sinon si elle est plus grande, on cherche dans l'arbre gauche, sinon on cherche dans l'arbre droit.

    On pourra écrire une fonction récursive, mais ce n'est pas obligatoire.

    Pour comparer les chaînes de caractères, on n'utilisera surtout pas `<` ou même `==`, qui comparent les adresses, mais on utilisera la fonction `strcmp` de la bibliothèque standard dans `string.h`, qui retourne `0` si les chaînes sont égales, une valeur strictement négative si la première est plus petite que la seconde, et une valeur strictement positive sinon.
13. Écrire une fonction `tree_add` qui prend en paramètre une clef, une valeur et un arbre, et qui retourne cet arbre dans lequel on aura rajouté l'association entre la clef et la valeur. L'arbre passé en paramètre sera potentiellement modifié, mais on le passera par valeur, puisque le nouvel arbre sera retourné par cette fonction.

    On utilisera pour cela l'algorithme récursif suivant :
    — si l'arbre est vide, on retourne un arbre avec un seul nœud contenant la clef et la valeur ;
    — si la clef de la racine de l'arbre est égale à la clef à ajouter, on remplace la valeur dans la racine ;
    — si elle est plus grande, on remplace le sous-arbre gauche par celui obtenu en y ajoutant récursivement l'association clef–valeur ;
    — si elle est plus petite, on remplace le sous-arbre droit par celui obtenu en y ajoutant récursivement l'association clef–valeur.

En ajoutant l'association `"chameau"` $\mapsto$ `8` dans l'exemple ci-dessus, on aura donc :



14. En déduire `map_add` qui modifie un dictionnaire par effet.
15. Pour `map_print`, on affichera d'abord récursivement le sous-arbre gauche, puis l'association à la racine, puis le sous-arbre droit. Attention aux arbres vides !

Les questions suivantes sont plus difficiles. Il est conseillé de passer aux autres parties et de ne revenir ici qu'à la fin. (`map_remove` n'est de toute façon pas utile pour la suite.)

16. Écrire une fonction `remove_root` qui prend en paramètre un arbre et qui retourne cet arbre dans lequel on a retiré la racine.

    Pour cela (n'hésitez pas à faire des dessins) :

    — Si le sous-arbre gauche est vide, on retourne le sous-arbre droit.

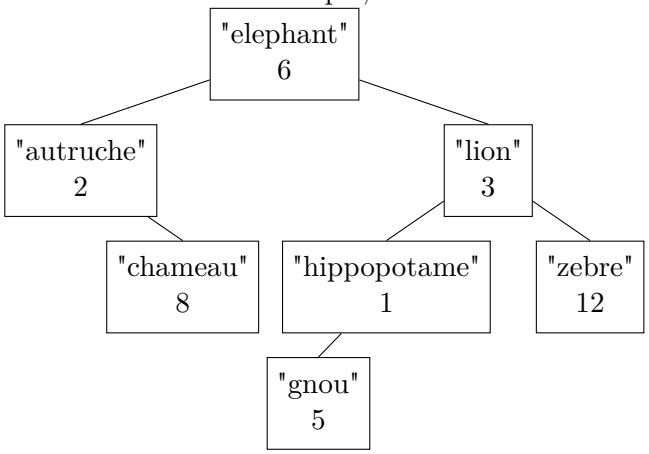
    — Si le sous-arbre droit est vide, on retourne le sous-arbre gauche.

    — Si le sous-arbre droit du sous-arbre gauche est vide, on le remplace par le sous-arbre droit de l'arbre initial et on retourne le sous-arbre gauche de l'arbre initial.

    — Sinon, on cherche le nœud le plus à droite du sous-arbre gauche (qui ne sera donc pas à sa racine), on remplace la clef et la valeur de la racine par celles de ce nœud, puis on remplace ce nœud par son sous-arbre gauche.

    On n'oubliera pas de libérer la mémoire du nœud qui n'est plus utilisé.

    En continuant sur l'exemple, on obtiendrait :



17. Écrire une fonction `tree_remove` qui prend en paramètre une clef et un arbre, et qui retourne cet arbre dans lequel on aura retiré l'association avec cette la clef si elle existe. L'arbre passé en paramètre sera potentiellement modifié, mais on le passera par valeur, puisque le nouvel arbre sera retourné par cette fonction.

    On utilisera pour cela l'algorithme récursif suivant :

    — si l'arbre est vide, on retourne l'arbre vide ;

    — si la clef de la racine de l'arbre est égale à la clef à retirer, on utilise `remove_root` ;

    — si elle est plus grande, on remplace le sous-arbre gauche par celui obtenu en y supprimant récursivement la clef ;

    — si elle est plus petite, on remplace le sous-arbre droit par celui obtenu en y retirant récursivement la clef.

18. En déduire `map_remove` qui modifie un dictionnaire par effet.

Exercice 3 : Utilisation : résolution de noms de domaine

On va utiliser les dictionnaires pour implémenter une version très simplifiée de la résolution de noms de domaine. Celle-ci consiste, étant donné un nom de domaine comme `www.google.com` ou `data.education.gouv.fr`, à obtenir l'adresse IP correspondante.

Pour simplifier, les adresses IP seront ici des entiers (type `int`). Le réseau Internet sera représenté par un tableau de dictionnaire. Les indices de ce tableau seront les adresses IP, et le dictionnaire situé à une case i sera le serveur de nom situé à l'adresse IP i : en lui demandant un nom de domaine n, il répondra une adresse IP j, à condition qu'il y ait l'association $n \mapsto j$ dans le dictionnaire.

Pour résoudre un nom de domaine $w_n.w_{n-1}.\cdots.w_0$, on procédera comme suit :

— on interroge le serveur racine (dans notre cas, celui à l'IP `0`) pour connaître l'adresse du serveur de noms de domaine pour le nom de domaine de premier niveau w_0 ;

— on interroge ce serveur pour connaître l'adresse du serveur pour le nom de domaine $w_1.w_0$;

— ainsi de suite jusqu'à demander l'adresse de $w_n.w_{n-1}.\cdots.w_0$.

Par exemple, supposons qu'on a le tableau suivant :

0	$\{$`"com"` $\mapsto 1$; `"fr"` $\mapsto 2\}$
1	$\{$`"google.com"` $\mapsto 3\}$
2	$\{$`"ensiie.fr"` $\mapsto 4$; `"gouv.fr"` $\mapsto 3\}$
3	$\{$`"www.google.com"` $\mapsto 5$; `"education.gouv.fr"` $\mapsto 8\}$
4	$\{$`"www.ensiie.fr"` $\mapsto 7$; `"aurionweb.ensiie.fr"` $\mapsto 6\}$

Si on cherche `www.ensiie.fr`, on interroge le serveur 0 avec `"fr"`, il nous répond l'IP 2 ; on interroge le serveur 2 avec `"ensiie.fr"`, il nous répond l'IP 4 ; on interroge le serveur 4 avec `"www.ensiie.fr"`, il nous répond 7. L'IP recherchée est donc 7.

Si on cherche `fr.wikipedia.org`, on interroge le serveur 0 avec `"org"`, il répond `-1`. Le nom de domaine est donc inconnu.

Si on cherche `www.amazon.com`, on interroge le serveur 0 avec `"com"`, il répond 1 ; on interroge le serveur 1 avec `"amazon.com"`, il répond `-1`. Le nom de domaine est donc inconnu.

Compléter le fichier `dns.c` :

19. Écrire une fonction `resolve` qui prend en paramètre une chaîne de caractères et un tableau de dictionnaires, et qui retourne l'IP associée à la chaîne (ou `-1` si elle est inconnue). Pour cela, on utilisera l'algorithme suivant :
    - On parcourt la chaîne de caractères, en empilant tous les suffixes situés après un point.

      Par exemple, pour `data.education.gouv.fr`, on aura au final une pile de la forme

`"fr"`
`"gouv.fr"`
`"education.gouv.fr"`
`"data.education.gouv.fr"`

      Remarque : le suffixe d'une chaîne `s` à partir de la position `i` peut être obtenu simplement avec `&s[i]`.
    - Tant que la pile n'est pas vide, et en commençant par le serveur d'IP 0, on interroge le serveur avec le nom de domaine au sommet de la pile ; s'il retourne `-1`, le nom de domaine est inconnu ; sinon, on continue en interrogeant le serveur à l'IP obtenue.
    - On retourne la dernière IP obtenue.
20. Compléter la fonction `main` pour fonctionner de la façon suivante : pour chacun des arguments passés en ligne de commande, on donne l'adresse IP correspondante, ou indique que l'adresse est inconnue.

    Exemple :

```
$ ./dns_resolve fr.wikipedia.org www.ensiie.fr www.amazon.com
IP of domain name fr.wikipedia.org could not be found.
IP of domain name www.ensiie.fr is 7.
IP of domain name www.amazon.com could not be found.
```