Mini-projet 2 (Modules et Généricité)

Objectifs

— Définir un type abstrait de données (TAD)
— Proposer plusieurs implantations d'une même structure de données
— Mettre en pratique tous les concepts vus

1 Consignes

1. Ce sujet constitue le **(mini-)projet 2**.
2. Ce mini-projet est un travail **individuel**.
3. Les programmes doivent fonctionner sur les machines Linux de l'ENSEEIHT et s'exécuter sans erreur signalées par valgrind.
4. Date limite pour rendre l'exercice 1 complet (via Git/Gitlab) : **samedi 16 novembre**.
   Date limite pour rendre l'ensemble du projet (via Git/Gitlab) : **samedi 30 novembre**.
   N'attendez pas le dernier moment !
5. Pour vous faciliter la tâche, l'interface et le programme de test de LCA sont fournis ainsi que le squelette du corps de ce module. L'interface et le programme de test pourront largement être réutilisés pour le module TH car LCA et TH sont des SDA.
6. Que d'acronymes ! Lisons vite le reste du sujet...

2 Travail à réaliser

On appelle structure de données associative [1] une structure de données qui permet d'enregistrer et rechercher des valeurs en fonction d'une clé identifiant de manière unique une valeur. Un exemple classique d'une telle structure de données est le dictionnaire. La clé est un mot et la valeur sa définition. Un autre exemple est un annuaire où la clé est le nom d'un abonné et la valeur les informations le concernant (numéro de téléphone, adresse, etc.).

Une structure de données associative (SDA) fournit les opérations suivantes.

1. Initialiser une SDA. La SDA ainsi initialisée est vide.
2. Détruire une SDA. Elle ne pourra plus être utilisée sauf à être initialisée à nouveau.

1. Cette structure de données correspond à la notion de *dictionnaire* (dict) en Python.

3. Enregistrer une valeur en précisant sa clé. Si la clé n'existe pas dans la SDA, elle est ajoutée avec la valeur associée. Si la clé existe dans la SDA, sa valeur est remplacée par la nouvelle. Après l'opération on a donc la valeur associée à la clé.
4. Supprimer la valeur associée à une clé dans une SDA. Cette clé n'existera plus dans cette SDA à l'issue de cette opération. L'exception `Cle_Absente_Exception` signale que la clé était absente de la SDA.
5. Savoir si une clé est présente dans une SDA.
6. Obtenir la valeur associée à une clé. L'exception `Cle_Absente_Exception` signale que la clé était absente de la SDA.
7. Obtenir la taille d'une SDA (le nombre de couples clé/valeur enregistrés).
8. Afficher l'état interne de la SDA. Cette opération est définie à fin de mise au point.
9. Appliquer un traitement pour chaque couple clé/valeur d'une SDA. Si le traitement échoue sur certains couples (levée d'une exception), l'opération devra quand même se continuer sur les couples suivants.

Dans la suite, nous envisageons plusieurs manières de réaliser une telle structure de données associative (type SDA et opérations associées). Nous choisissons d'adopter une approche de type **programmation défensive**. Dans les exemples, nous considérerons que la clé est une chaîne [2] de caractères et la valeur un entier mais les **modules** qui seront écrits doivent être **génériques**.

Exercice 1 : Liste Chaînée Associative

Nous appelons *Liste Chaînée Associative* (LCA) cette première réalisation d'une structure de données associative car elle s'appuie sur une liste chaînée linéaire simple où chaque cellule contient une clé, une valeur et un accès à la cellule suivante.

1.1. Lire et comprendre l'interface du module générique LCA (fichier `lca.ads`).

1.2. Écrire un programme minimal (fichier `lca_sujet.adb`) qui insère successivement dans une même LCA la valeur 1 associée à la clé "un" et la valeur 2 associée à la clé "deux" et affiche cette LCA au moyen de l'opération numero 8 pour visualiser la structure de la LCA. L'affichage ressemblera à `-->["un" : 1]-->["deux" : 2]--E`.

1.3. Écrire l'implantation du module LCA (fichier `lca.adb`). On pourra se demander s'il ne serait pas judicieux de définir de nouveaux sous-programmes pour faciliter l'implantation des sous-programmes de l'interface.

1.4. Indiquer les inconvénients/avantages d'une implantation par structures chaînées d'une SDA. On répondra à cette question dans le fichier `LISEZ-MOI.txt`.

1.5. Lire et comprendre le programme de test (fichier `test_lca.adb`) du module LCA. L'utiliser pour tester votre module LCA.

Exercice 2 : Tables de hachage

Pour améliorer le temps d'exécution, on peut s'appuyer sur le principe des *tables de hachage*

2. En Ada, on utilisera le type `Unbounded_String` du module `Ada.Strings.Unbounded`. Ce sont des chaînes de taille variable. Deux fonctions `To_String` et `To_Unbounded_String` permettent de passer d'une `Unbounded_String` à une `String` et inversement. Le programme `exemple_unbounded_string.adb` montre l'utilisation des `Unbounded_String` qui sont aussi utilisées dans le programme de test fourni.

(TH) pour réaliser notre SDA. L'idée est d'utiliser 1) un tableau (donc un accès direct) pour stocker les valeurs et 2) une fonction, appelée *fonction de hachage*, qui à partir d'une clé calcule la position de la valeur dans le tableau. La position calculée par une fonction de hachage est appelée *valeur de hachage*.

La fonction de hachage pourrait être la fonction qui renvoie le nombre de caractères de la clé, le code ASCII de l'initiale [3] de la clé, la somme des codes ASCII de ses caractères, etc.

Par exemple, si la fonction de hachage est la longueur de la clé, si le tableau est indicé de 0 à 10, la clé "deux" a une longueur de 4 et ira donc dans la case d'indice 4 du tableau. La clé "quatre", de valeur de hachage 6, ira dans la case d'indice 6.

Cette approche pose deux problèmes :

1. La valeur de hachage peut être trop grande par rapport à la taille de la table de hachage (i.e. la capacité du tableau). Par exemple la clé "quatre-vingt-dix-neuf" a pour valeur de hachage 21 qui est au-delà des indices valides sur le tableau. La solution est alors d'utiliser le reste de la division entière pour trouver l'indice. Ainsi, la taille de notre table étant 11 (indices valides de 0 à 10), l'indice sera 21 modulo 11, c'est-à-dire 10.
2. Deux clés différentes peuvent avoir la même valeur de hachage. Par exemple la valeur de hachage de la clé "cinq" est 4 comme la clé "deux". On dit qu'il y a collision. Une solution pour gérer ces collisions est de considérer que chaque case de notre tableau est en fait une liste chaînée associative (exercice 1). Ainsi on pourra y insérer plusieurs couples clé/valeur.

L'utilisateur de la table connait les informations qu'il souhaite mettre dedans. Aussi, il est le plus à même de définir la taille et la fonction de hachage qui minimisent les collisions.

2.1. Écrire l'interface du module TH (fichier `th.ads`). Notons que comme c'est aussi une SDA, l'interface du module TH est très proche de celle du module LCA.

2.2. Écrire un programme (fichier `th_sujet.adb`) qui crée une table de hachage de taille 11, dont les clés sont des chaînes, les valeurs des entiers et la fonction de hachage la longueur de la clé. Ce programme enregistrera ensuite dans cette table les valeurs entières 1, 2, 3, 4, 5, 99 et 21 avec les clés respectives "un", "deux", "trois", "quatre", "cinq", "quatre-vingt-dix-neuf" et "vingt-et-un". Enfin, il affichera le contenu de la table grâce à l'opération numero 8. L'affichage ressemblera à

```
 0 -->["vingt-et-un" : 21]--E
 1 --E
 2 -->["un" : 1]--E
 3 --E
 4 -->["deux" : 2]-->["cinq" : 5]--E
 5 -->["trois" : 3]--E
 6 -->["quatre" : 4]--E
 7 --E
 8 --E
 9 --E
10 -->["quatre-vingt-dix-neuf" : 99]--E
```

2.3. Écrire l'implantation du module TH (fichier `th.adb`).

3. On retrouve alors le principe des répertoires papier.

2.4. Écrire un programme de test du module TH (fichier `test_th.adb`). On pourra s'inspirer du programme de test du module LCA (question 1.5) : il suffira de créer une TH au lieu d'une LCA.

Exercice 3 : Exploitation des SDA

Pour comparer expérimentalement les performances de nos deux SDA, LCA et TH, nous allons écrire un programme qui mesure la qualité d'un générateur de nombre aléatoire. Le principe du programme sera de mesurer la fréquence d'apparition des nombres d'un échantillon de nombres tirés aléatoirement dans un intervalle `1..Borne` et d'afficher la plus petite fréquence et la plus grande fréquence. Le générateur aléatoire est de qualité si la différence entre ces deux fréquences est faible. Bien sûr, il faut un échantillon de grande taille par rapport à la valeur de Borne pour que les résultats soient pertinents.

Ce programme sera décliné suivant les deux SDA : `evaluer_alea_lca` et `evaluer_alea_th`. Seule la définition de la SDA à utiliser changera d'un programme à l'autre, LCA pour le premier, TH pour le second. Chacun de ces programmes prendra en argument, sur la ligne de commande, d'abord la valeur de *Borne* puis la taille de l'échantillon, *Taille*. Le programme affichera la borne, la taille et les deux fréquences comme illustré ci-après.

```
> ./evaluer_alea_th 1000 10000000
Borne  : 1000
Taille : 10000000
Min : 9674
Max : 10376
```

Pour le programme avec la table de hachage, on fixera la taille de la table à 1000 et on prendra comme fonction de hachage la fonction identité qui à N associe N.

On pourra faire différents essais avec Borne valant 10, 100, 1000, 10000 et 100000 et Taille variant 10 à 10^7 (en considérant les puissances de 10).

Attention ces programmes devront être robustes !

Un squelette est donné pour les fichiers `evaluer_alea_lca.adb` et `evaluer_alea_th.adb`. Il montre comment exploiter les arguments de la ligne de commande. Attention, les programmes fournis ne sont pas robustes. On peut par exemple le constater en faisant :

```
> ./evaluer_alea_lca 10 xyz
```

Pour mesurer le temps d'exécution sous Unix, on peut utiliser la commande `time` comme par exemple :

```
> time ./evaluer_alea_th 1000 10000
Borne  : 1000
Taille : 10000
Min : 2
Max : 23

real 0m0,056s
user 0m0,047s
sys 0m0,009
```

Il s'agit alors de regarder la valeur correspondant au temps utilisateur (*user*).

Prenons un autre exemple : la commande `sleep` qui fait une attente du nombre de secondes indiqué. On constate que le temps utilisateur est très différent du temps réel. Le programme a peu utilisé de temps CPU (*user*) même s'il a mis environ 10 secondes pour se terminer (*real*).

```
> time sleep 10

real 0m10,03s
user 0m0,001s
sys 0m0,002s
```