TP 5

Dans le cadre de l'UE, vous êtes amenés à programmer en **C++** en utilisant les références pour réaliser vos passages de paramètres, les constructeurs, les destructeurs et les surcharges de l'opérateur d'affectation. Néanmoins, il sera important que vous sachiez toujours basculer dans un autre langage de programmation. Pour cela, il est important que vous établissiez votre raisonnement au niveau du pseudo-langage algorithmique, en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

1 Tables de hachage

Développez un module (`table.h`, `table.cpp`) offrant le type abstrait `Table`, en vous appuyant sur une implantation de type `Table de hachage`. Ces `Table`s seront utilisées pour ranger les informations relatives à des produits proposés par une petite épicerie qui est limitée dans le nombre «`m` de produits qu'elle peut vendre. Le type `clé` correspond ici au numéro de produit codé dans un `unsigned int`, et l'`information associée` est un prix codé dans un double. Dans un souci de modularité, prévoyez tout de même de pouvoir utiliser votre module avec des clés et des informations associées d'un autre type. Cela signifie que vous développerez un module `Clé` et un module `InfoAssociée` utilisés par le module `Table`.

Bien entendu le nombre de valeurs possibles d'une `clé` est beaucoup plus grand que «`m` et les `clé`s ne peuvent donc pas être utilisées pour indicer directement les cases d'un tableau dans lesquelles on consignerait les prix. Une `Table de hachage` de taille maximale largement inférieure à `m` sera néanmoins implantée avec un tableau de taille fixe `m` contenant des clés et des informations associées.

Dans une `Table de hachage` de taille `m`, l'accès à l'emplacement d'une `clé` (et à son `information associée`) se fait par application d'une fonction de hachage sur la `clé` pour obtenir l'indice de sa place présumée dans le tableau de **taille fixe** `m` modélisant la `Table`.

Votre table de hachage sera donc caractérisée par une capacité maximale `m` fixée à l'initialisation (paramètre du constructeur) et vous effectuerez une gestion des collisions par adressage ouvert (re-hachage).

- Prévoyez la possibilité pour l'utilisateur de votre module de pouvoir fixer la **fonction de hachage de son choix** (fonction retournant un entier inférieur à `m` à partir d'une `clé`) au moment de l'initialisation de la table. Il s'agit donc d'un second paramètre du constructeur. Vous pourrez ainsi comparer les différentes fonctions de hachage vues en cours.
- Prévoyez également la possibilité d'opter pour un re-hachage linéaire, un re-hachage quadratique ou un double-hachage. En pratique, vous mettrez ceci en œuvre en fournissant une seconde fonction (dernier paramètre du constructeur) qui donne le pas de rehachage, c'est à dire l'intervalle en nombre de cases entre deux essais consécutifs (ie entre le i+1 ème essai et le précédent).
- Votre type abstrait devra offrir une opération de `suppression` de l'entrée correspondant à une `clé`, en plus des traditionnelles opérations d'`insertion` et de `recherche` d'une `clé`.
- Prévoyez enfin une fonction d'`affichage` de l'état interne de la table : affichage de la `clé` contenue dans une case si elle est non vide, ainsi que le nombre d'essais réalisés pour insérer cette `clé`.
- Bien entendu, vous offrirez également des fonctions permettant de retourner ou de modifier l'`information associée` à une `clé`.
- N'oubliez pas de stocker le nombre d'éléments `n` stockés dans la table. Attention, l'utilisateur d'une table devra veiller à ce que `n` reste inférieur à `m`. Aucun aggrandissement de la

table n'est possible.

Au niveau de la mise en œuvre, il sera particulièrement important que le choix de la `fonction de hachage`, ainsi que le choix de la fonction fournissant le `pas de re-hachage` (intervalle séparant les indices des différents essais d'adressage) soit géré à travers l'utilisation de **pointeurs de fonction**, pour permettre de faire des comparaisons de performance sur la base d'un même programme, dans lequel on injecte des stratégies différentes (Attention, suivant le rehachage choisi, le `pas de re-hachage` peut dépendre du nombre d'essais déjà réalisés pour insérer ou rechercher la clé et/ou de la valeur de la clé).

2 Performances

Réalisez une évaluation de performances, de manière à obtenir le profil du temps de recherche d'une `clé` en fonction du nombre `n` <`m` d'éléments présents dans la table.

Vous procéderez à une mesure des performances de la procédure de localisation d'une `clé` en fonction du nombre `n` <`m` de `clé`s présentes dans la table (les `clé`s utilisées par votre programme pourront être générées aléatoirement, à la volée). Comme pour les mesures de performances réalisées au TP précédent, vous calculerez des moyennes du temps de recherche d'une `clé` en travaillant simultanément sur plusieurs `Table`s. Vos performances (moyenne du temps nécessaire à la localisation d'une `clé` dans une table en contenant `n`) seront enregistrées dans un fichier `performance.txt`. Comme suggéré au TP précédent, vous pourrez utiliser la classe `std::chrono` pour faire vos mesures de temps.

Exemple de fichier `performance.txt` :

```
# "nb" "Temps"
1000 4.5
2000 5.1
3000 8.9
5000 10.7
```

Les fichiers ainsi écrits pourront être lus avec `gnuplot` de manière à visualiser les temps d'exécution sur un diagramme.

On comparera les performances dans le cas d'un rehachage linéaire, quadratique (vérifiez l'avantage de prendre `m` premier) ou un double hachage (vérifiez l'avantage de prendre `m` correspondant à une puissance de 2).