TP 2 (prélude du TP3)

Dans le cadre de l'UE, vous êtes amenés à programmer en **C++** en utilisant les références pour réaliser vos passages de paramètres (MODIFIABLES ou NON MODIFIABLES), 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 et en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

1 Les listes d'aujourd'hui sont triées

1.1 Sentinelle

— Commencez par reprendre votre implémentation C++ des listes chaînées (non circulaires) et modifiez la de manière à la munir d'une `Cellule` **bidon/sentinelle** (dont le champs info n'est pas pertinent), avant la première `Cellule`. La liste vide est donc munie de cette `Cellule` sentinelle. Attention, à cette étape on vous demande seulement de modifier l'implémentation de la liste, pas son interface.
L'utilisation d'une `Cellule` sentinelle vous était également demandée dans votre implémentation des listes chaînées circulaires. Avez-vous compris l'utilité d'une telle cellule ?

— Dans un programme utilisateur, créez une **liste triée croissante** par **ajout** successif de **valeurs décroissantes**.

1.2 Module Liste triée

— Comment modifier l'interface du module `Liste` pour le transformer en un module `Liste triéee` ? Commencer à apporter vos modifications dans le fichier `ListeTriee.h` avant de vous lancer dans le fichier d'implémentation `ListeTriee.cpp`. En particulier, vous munirez votre module d'une fonction membre `insere` qui permettra d'ajouter un élément dans une `Liste triée`. Quel sera selon-vous son prototype ? Dans un premier temps on ne demande pas de munir votre module de fonctionnalités de recherche ou de suppression d'un élément. Prévoir en revanche les classiques opérations d'initialisation (constructeurs), affectation (par surcharge d'opérateur), testament (destructeur).

— Testez les fonctionnalités de votre module `Liste triéee` dans un programme de test au fur et à mesure que vous les développez.

2 Ajoutons un étage

— Dans la liste actuelle, les `Cellule`s contiennent un unique pointeur qui stocke l'adresse de la `Cellule` suivante dans la liste ou *nullptr* s'il n'y a pas de `Cellule` suivante. On se propose de modifier la structure de la `Cellule` en ajoutant un deuxième pointeur qui stockera l'adresse de la `Cellule` située deux `Cellule`s plus loin si elle existe. Ceci pour une `Cellule` sur deux de la liste, le second niveau de chainage étant nul pour les autres `Cellule`s. On modifiera également la structure de `Liste triée` en ajoutant un booléen qui permettra de savoir si le deuxième niveau de chaînage a été mis en place ou pas (les pointeurs intervenant dans le second niveau de chaînage étant tous initialisés à *nullptr*) (voir figure).

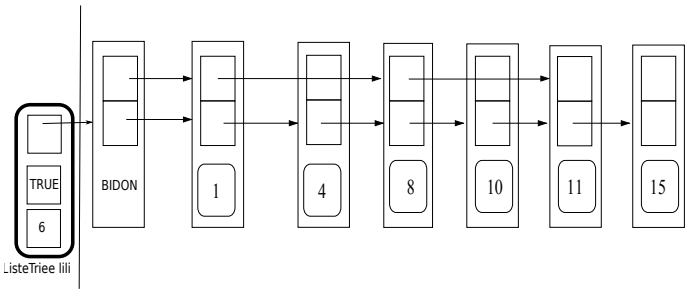

FIGURE 1 – Exemple de `Liste triée` utilisant une `Cellule` bidon (sentinelle) et un second niveau de chaînage.

— Ecrivez une fonction membre `etablissementSecondNiveau` modifiant une `Liste triéee` pour la munir du second niveau de chaînage. Pour une liste non vide, on imposera que les 2 pointeurs de la `Cellule` bidon/sentinelle soient non nuls. Ils pointent respectivement vers la première `Cellule` (non sentinelle) du niveau 1 et du niveau 2. Dans un premier temps, on conviendra que cette première `Cellule` (non sentinelle) du niveau 1 est également la première `Cellule` du niveau 2.
— Munissez la `Liste triée` d'une seconde fonction membre d'affichage des éléments présents sur le second niveau de chaînage.
— Si vous utilisez votre fonction membre actuelle d'insertion d'un élément dans une `Liste triéee`, vous pouvez constater qu'elle fonctionne toujours. Quelle observation pouvez-vous cependant faire ? A cette étape, vous n'avez pas à modifier cette fonction.

3 Quand le hasard s'en mêle

— Dans la partie précédente, il y avait deux `Cellule`s d'écart entre deux `Cellule`s consécutives du second niveau de chaînage.
— On propose à présent que l'écart entre deux `Cellule`s consécutives devienne aléatoire. Pour cela, lorsqu'on met en place le second niveau de chaînage, on parcourt l'ensemble des `Cellule`s du premier niveau et, pour chacune, on tire à pile ou face pour décider si elle sera retenue dans le second niveau ou pas. Modifiez votre fonction membre `etablissementSecondNiveau` pour prendre en compte cet aspect aléatoire. Cette fois-ci la première `Cellule` (non sentinelle) du niveau 1 n'est pas forcément la première `Cellule` du niveau 2.
— Avez-vous une idée pour que la fonction d'insertion d'un élément dans une `Liste triéee` exploite le second niveau de chaînage ? Modifiez votre fonction pour la rendre plus efficace. Pour chacun des éléments insérés, on tirera au sort s'il doit être inséré dans les deux niveaux ou seulement dans le niveau de base.
— Voici à présent un exemple de programme utilisateur.

```
ListeTriee lili;
for(i=0;i<10;i++)
   lili.insere(std::rand()%100)// #include <cstdlib>
lili.affiche();
lili.etablissementSecondNiveau();
lili.affiche(); // Meme affichage que precedemment
lili.afficheNiveau2();
for(i=0;i<10;i++)
   lili.insere(std::rand()%100); //Cette fois insere peut se servir
                                 //du second niveau de chainage
lili.affiche();
lili.afficheNiveau2();
```