

## 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, en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

### 1 Sentinelle

L'utilisation d'une **Cellule** sentinelle était demandée au TP01 dans l'implémentation des listes chaînées circulaires. Aviez-vous compris l'utilité d'une telle cellule dans le contexte circulaire ?

Rajoutons une sentinelle dans le contexte non circulaire :

- 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**. Même la liste vide est munie de cette **Cellule** sentinelle. Attention, on vous demande seulement de modifier l'implémentation de la liste, pas son interface. Trouvez vous un intérêt à l'ajout de cette cellule quand on est dans un contexte non circulaire ?
- Dans un programme utilisateur, créez une **liste triée croissante** par **ajout** successif en début de liste de **valeurs décroissantes** et affichez la.

### 2 Module Liste triée

- Comment modifier l'interface du module **Liste** pour le transformer en un module **Liste triée** ? Commencer à apporter vos modifications dans le fichier `ListeTrie.h` avant de vous lancer dans le fichier d'implémentation `ListeTrie.cpp`. Des fonctionnalités vont disparaître au profit de nouvelles. En particulier, vous munirez votre module d'une fonction membre `insere` qui permettra d'ajouter un élément à sa place 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), testement (destructeur).
- Testez les fonctionnalités de votre module **Liste triée** dans un programme de test au fur et à mesure que vous les développez.

### 3 Ajoutons un étage

- Dans la liste actuelle, les **Cellules** 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. Pour la cellule bidon et pour une cellule sur deux de la liste, on utilisera ce deuxième pointeur pour stocker l'adresse de la **Cellule** située deux **Cellules** plus loin dans la liste si elle existe. Pour les autres **Cellules** de la liste, ce second niveau de chaînage restera nul (voir figure). 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`).

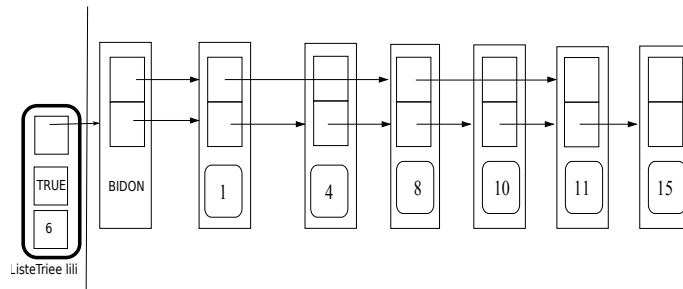


FIGURE 1 – Exemple de `Liste triée` utilisant une `Cellule` bidon (sentinelle) et un second niveau de chaînage.

- Ecrivez une fonction membre `etablissementSecondNiveau` modifiant une `Liste triée` pour la munir du second niveau de chaînage. Pour une liste non vide, les 2 pointeurs de la `Cellule` bidon/sentinelle pointent respectivement vers la première `Cellule` (non sentinelle) du niveau 1 et 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 triée`, vous pouvez constater qu'elle fonctionne toujours. Quelle observation pouvez-vous cependant faire ? A cette étape, vous n'avez pas à modifier cette fonction, mais pensez vous que le second niveau de chaînage pourrait améliorer sa complexité ?

## 4 Quand le hasard s'en mêle

- Dans la partie précédente, il y avait deux `Cellules` d'écart entre deux `Cellules` consécutives du second niveau de chaînage.
- On propose à présent que l'écart entre deux `Cellules` consécutives devienne aléatoire. Pour cela, lorsqu'on met en place le second niveau de chaînage, on parcourt l'ensemble des `Cellules` 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 triée` 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.etablissementSecondNiveau();
lili.affiche();
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();
```