Représentation générique d'un graphe

Introduction

L'objectif de ce TP est la mise en oeuvre générique (en C++) de la notion de graphe. La solution retenue doit être indépendante de la nature de la problématique représentée par le graphe. Elle doit permettre la mise en oeuvre des algorithmes d'exploration classiques et la mise en oeuvre de l'algorithme du recuit simulé appliqué à la résolution du problème du voyageur de commerce.

La décomposition du problème nous conduit à définir les étapes suivantes :

1. Mise en oeuvre d'une classe liste récursive générique appelée *PElement*. Cette classe est la brique de base. Elle va permettre de représenter aussi bien un graphe, une liste d'adjacences que la liste des sommets ouverts d'un algorithme d'exploration comme A*.
2. Mise en oeuvre des classes génériques *Sommet*, *Arete* et *Graphe*. L'association de ces trois classes permet de représenter un graphe non orienté quelconque.
3. Pour vérifier que notre définition de graphe est fonctionnelle, nous choisissons un exemple d'application : représentation d'une carte routière simplifiée.
   Ceci nous amène à définir les classes suivantes :
   - classe *InfoSommetCarte* : contient les informations relatives à un lieu d'une carte routière
   - classe *InfoAreteCarte* : contient les informations relatives à une arête, nécessaires au déroulement d'un algorithme d'exploration
   - classe *DessinGraphe* et *DessinGrapheRecuitSimule*: permettent de dessiner une carte routière par l'intermédiaire d'un fichier texte à l'aide de l'appli JAVA *bsplines*.
   - classe *OutilsCarteRecuitSimule*, qui contient quelques méthodes utiles à la création d'une carte routière simplifiée à l'aide d'un *Graphe*.

4. Finalement construction puis dessin, dans la fonction main, d'une carte routière simplifiée

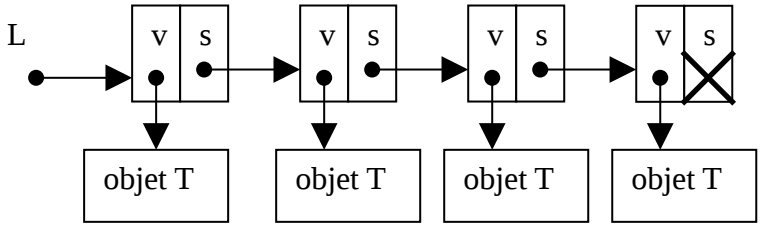
 **Remarque :**

Dans cette approche, la généricité est assurée à l'aide de la notion de *template* propre au langage C++. Ce concept est, dans ce contexte, bien plus adéquat que la notion de fonction virtuelle. Cette dernière est, en effet, adaptée à représenter des listes *hétérogènes* (deux éléments consécutifs ne sont pas de même nature) qui n'apparaissent pas dans ce problème.
A la complexité et au coût en temps de la solution "fonctions virtuelles", nous préférons donc la solution *template* plus simple et plus efficace.

TP1. Mise en oeuvre de la classe liste récursive générique PElement<T> * (1h30)

1. Définir la classe PElement<T> générique

Un maillon de la liste chaînée est de type *PElement<T>*. Il contient 2 attributs :
un lien vers le maillon suivant; ce lien est noté *s*.
un lien vers la valeur (c'est-à-dire l'information associée à ce maillon). Ce lien est noté *v*.
v est un pointeur vers une instance d'une classe *T* générique.
Pour simplifier, *v* et *s* peuvent être publics.
Une liste chaînée L est obtenue en prenant un pointeur sur un *PElement<T>*.
Une liste *PElement<T>* * L a donc en mémoire, la forme suivante:



Comme toujours, l'attribut *s* du dernier maillon de la liste vaut NULL. De même, une liste vide est représentée par NULL.
Munir *PElement<T>* d'un constructeur.
Ce constructeur ne fait pas de copie de ses arguments (pas de new, pas de malloc).
Destructeur, getters et setters sont inutiles.

☞ **Attention** ! Avec une solution template, il n'y a pas de fichier .cpp. Tout le code est donc écrit dans le fichier PElement.h.
Ecrire un fichier source *TestPElement.cpp* contenant une fonction *main*() qui teste les fonctionnalité de la classe PElement<T>.
Dans main(), créer deux listes chaînées l1 et l2. l1 comme suit :

```
PElement<double> * l1;
PElement<string> * l2;
```

avec l1 = (2, 5, 7.5, 9)
et l2 = ("carotte","cerise","orange")

☞ **Astuce :** pour enchaîner un maillon en tête, ne pas oublier la forme d'appel suivante :

```
... = new PElement<T>(...,...)
```

2. Munir la classe PElement<T> de la méthode :

```
static int taille(const PElement<T> * l);
```

qui calcule la taille de la liste l (la solution récursive est plus simple).
La tester sur l1 et l2.

3. Munir la classe PElement<T> de la méthode :

```
static const string toString(const PElement<T> * p,
                             const char * debut="( ",
                             const char * separateur = ", ",
                             const char * fin=")");
```

qui transforme la liste p en string (la solution avec une boucle est plus simple).
Le plus simple est d'utiliser un *ostringstream* intermédiaire.
On suppose que l'opérateur << d'écriture sur un flux est défini pour la classe T.
Tester *toString* sur l1 et l2.
Ecrire l'opérateur suivant comme fonction ordinaire (ni membre, ni friend) :

```
ostream & operator <<(ostream & os, const PElement<T> * p)
```

Idéalement, cet opérateur se ressert de *toString*.
Tester << sur l1 et l2.

4. Munir la classe PElement<T> de la méthode :

```
static void insertionOrdonnee(      T * a,
                                    PElement<T> * & l,
bool (*estPlusPetitOuEgal)(const T * a1, const T * a2));
```

qui insère en ordre la nouvelle valeur *a* dans la liste *l* en se servant de la méthode de comparaison *estPlusPetitOuEgal*. *l* peut être modifiée par l'appel (d'où le &).
La version récursive est, de loin, la plus simple.
On suppose que, avant l'appel, *l* est ordonnée par ordre croissant (l'ordre est défini par *estPlusPetitOuEgal*). Après l'appel, *l* est encore ordonnée.
Tester *insertionOrdonnee* en insérant successivement les valeurs -7, 6 et 13 dans l1 et les valeurs "fraise", "ananas" et "pomme" dans l2.
Ce test exige d'écrire deux méthodes de comparaison *double* à *double* et *string* à *string* (pour cette dernière méthode, penser à utiliser la méthode string::compare).

5. Munir la classe PElement<T> de la méthode :

```
static T * depiler(PElement<T> * & l);
```

qui retire l'élément situé en tête de *l* et le renvoie. Le premier maillon de *l* est effacé.
l est donc modifiée par l'appel. En sortie *l* compte un maillon de moins.
La méthode lance une exception de type *Erreur* si *l* vaut NULL à l'appel.
Tester *depiler* sur l1 et l2.

6. Munir la classe PElement<T> de la méthode :

```
static bool retire(const T * a, PElement<T> * & l);
```

qui retire la 1ère occurrence de *a* de *l* si *a* est présent dans *l*, sinon ne fait rien.
L'élément trouvé *v* n'est pas effacé mais le maillon le contenant est effacé.
La version récursive est nettement plus facile à écrire.
La comparaison est faite sur les pointeurs *v*.
Données : *a*, *l*
Résultats : *l* (éventuellement modifiée), par return : *true* si l'élément a été trouvé, *false* sinon.
Tester *retire* sur l1 et l2.

7. Munir la classe PElement<T> de la méthode :

```
static void efface1(PElement<T>* & l);
```

qui efface tous les maillons de la liste *l* mais qui n'efface pas les données *v. En sortie *l* vaut NULL. Ne fait rien si *l* vaut NULL à l'appel.
La version récursive est plus facile à écrire.

Tester *efface1* sur l1 et l2.

8. Munir la classe PElement<T> de la méthode :

```
static void efface2(PElement<T>* & l);
```

qui efface tous les maillons de la liste *l* et toutes les données *v. En sortie *l* vaut NULL. Ne fait rien si *l* vaut NULL à l'appel.
La version récursive est plus facile à écrire.

Tester *efface2* sur l1 et l2.