Université Grenoble Alpes Département INFO

R3.02 – Développement efficace

TP 1 (associé au Cours 1)

Listes chaînées (1 séance encadrée et travail personnel)

Pour ce TP

- Télécharger et compléter le projet CLion « R302-TP1 »
- On fournit une configuration d'exécution par exercice.

Liste chaînée simple

On propose la classe abstraite modèle **ListeInterface** (fichier **ListeInterface**.h) qui réalise le Type Abstrait de Données **Liste**.

On propose une implantation partielle de la classe modèle **ListeChainee** (fichiers **ListeChainee.h**) qui réalise le TAD Liste sous forme de liste de **Cellules**.

La classe **ListeChainée** proposée est fournie Annexe 1 : liste chaînée certaines méthodes seront implantées avec différents algorithmes d'où des noms de méthodes explicites sur le type d'algorithme.

Afin de faciliter la lecture de la Figure 3 : Spécification UML de la classe ListeChainee,

- les procédures et fonctions privées, qui font le travail, sont définies sur la ligne qui précède la spécification des méthodes publiques qui les utilisent.
- les **fonctions**, **procédures** et **méthodes** dont l'**implantation** est **fournie** sont notées dans la police **Courrier**.
- les fonctions, procédures et méthodes dont l'implantation est demandée sont notées dans la police Arial.

Note importante:

Veiller toujours à faire apparaître l'algorithme sous forme de commentaire après le prototype de la méthode à compléter.

Exercice 1 : La liste chaînée

I. Suppression à une certaine position (récursif)

 Observer la méthode supprimeAtPositRec(const int position) de ListeChainee, implanter la procédure supprimeAtPositRecWorker(), et la tester avec testeSupprimeAtPositRec() dans TesteListe.cpp.

Note Les méthodes supprimeTete() et supprimeTeteWorker(Cellule<TypeInfo>*& ptrCetteListe) sont fournies.

II. Suppression à une certaine position (itératif)

 Implanter la méthode supprimeAtPositIter() de ListeChainee, et la tester avec testeSupprimeAtPositIter() de exercice 2.

III. Insertion à une certaine position (itératif)

 Implanter la méthode insereAtPositIter(int position, const TypeInfo& nouvelleInfo)de ListeChainee, et la tester avec testeInsereAtPositIter() de exercice 2.

R3.02—TP1 Page 1 sur 8

IV. Une valeur est-elle présente dans une liste

La méthode estInfoPresenteRec est fournie.

 Implanter la méthode bool estInfoPresentRecWorker(const Cellule<TypeInfo>* ptrCetteListe, const TypeInfo& infoCible) de ListeChainee, et la tester avec testeEstInfoPresentRec() de exercice 2.

Exercice 2 : Le TAD pile et son utilisation (à faire en dernier)

Une **pile** est une collection linéaire d'éléments où les consultations, les insertions, les suppressions se font toujours du même côté (dernier arrivé, premier servi). C'est une métaphore de la pile physique d'objets (exemple : la pile d'assiettes).

Le vocabulaire des opérations est basé sur une représentation verticale (comme une pile d'objets physique). Sur une pile on peut empiler un élément (au dessus du sommet), dépiler l'élément au sommet, vérifier si la pile est vide ou non.

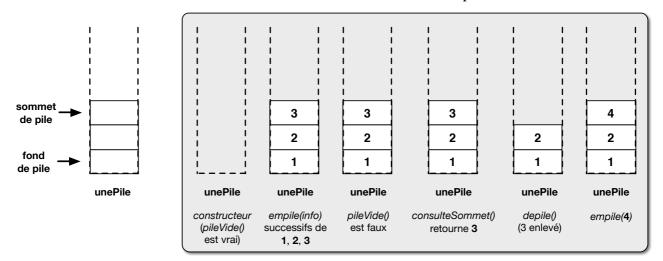


Figure 1: TAD pile, illustration

La spécification UML de la classe abstraite **PileInterface** qui définit la TAD pile est proposée Annexe 2 : pile. Une pile s'implémente facilement au moyen de **Cellules** ; réfléchir au pourquoi et au comment. On propose la spécification de la classe **PileCellules** en Annexe 2 : pile.

I. Implantation et test de la classe pile

- Compléter l'implantation de la classe **PileCellules** (méthodes depile() et empile()) dans le fichier PileCellules.h.
- Tester votre implantation avec exercice 2.

II. Affichage itératif de droite à gauche d'une ListeChainee

- Réfléchir à l'algorithme que l'on peut mettre en œuvre pour réaliser l'affichage d'une ListeChainee en réalisant un parcours de droite à gauche en utilisant une pile comme structure de données auxiliaire,
- Implanter la méthode suivante dans la classe ListeChainee.
 void afficheDGIter();
 // affichage de cette ListeChainee avec un parcours de droite à gauche
- Tester la méthode afficheDGIter() avec la procédure testeAfficheIter() de exercice 2.

Page 2 sur 8 R3.02—TP1

Liste chaînée triée croissante

On propose la classe abstraite modèle **ListeTrieeInterface** (fichier **ListeTrieInterface.h**) qui réalise le Type Abstrait de Données **ListeTriée**.

On propose une implantation partielle de la classe modèle ListeTrieeChainee (fichier ListeChainee.h) qui réalise le TAD Liste sous forme de liste de Cellules.

La classe **ListeChainée** proposée est fournie Annexe 3 : liste chaînée triée certaines méthodes seront implantées avec différents algorithmes d'où des noms de méthodes explicites sur le type d'algorithme.

Afin de faciliter la lecture de la Figure 6 : Spécification UML de la classe ListeChaineeTriee,

- les **procédures** et **fonctions privées**, qui font le travail, sont définies sur la ligne qui précède la spécification des méthodes publiques qui les utilisent.
- les **fonctions**, **procédures** et **méthodes** dont l'**implantation** est **fournie** sont notées dans la police **Courrier**.
- les fonctions, procédures et méthodes dont l'implantation est demandée sont notées dans la police Arial.

Note importante:

Veiller toujours à faire apparaître l'algorithme sous forme de commentaire après le prototype de la méthode à compléter.

Exercice 3 : Autour de la liste triée croissante

I. Insertion associative (récursif)

- Observer la méthode insereRec(TypeInfo& nouvelleInfo) de ListeChaineeTriee,
- Implanter la méthode

II. Position d'une information (itératif)

 Implanter la méthode int getPositIter(const TypeInfo& uneInfo) et la tester avec testeGetPositIter() de exercice 3.

III. Suppression associative (récursif)

- Observer la méthode bool supprimePremOccRec(const TypeInfo& uneInfo) de ListeChaineeTriee,
- Implanter la méthode

et la tester avec testeSupprimePremOccRec() dans TesteListeChaineeTriee.cpp.

IV. Suppression associative (itératif)

- Implanter la méthode

bool supprimePremOccIter(const TypeInfo& uneInfo)

et la tester avec testeSupprimePremOccIter() dans TesteListeChaineeTriee.cpp.

R3.02—TP1 Page 3 sur 8

V. Suis-je un ensemble ? (itératif)

```
Implanter la méthode
```

```
bool estEnsemble();
// retourne faux si au moins une info est présente plusieurs fois
// dans cette ListeChaineeTriee ; vrai sinon {c'est un ensemble trié}
et la tester avec testeEstEnsemble() de exercice 3.
```

VI. Suppression de toutes les occurrences multiples

- Implanter la méthode

```
ListeChaineeTriee<TypeInfo>* supprimeToutesDuplications();
// retourne une ListeChaineeTriee qui ne contient plus qu'une seule
// occurrence des informations présentes dans cette ListeChaineeTriee
// le résultat peut être considéré comme un ensemble trié
```

et la tester avec testeSupprimeToutesDuplications() de exercice 3.

VII. Union de deux ensembles triés

- Ajouter, en l'implantant, la méthode suivante :

```
ListeChaineeTriee<TypeInfo>*
```

```
unionAvec(ListeChaineeTriee<TypeInfo>* ensembleB);
// retourne une ListeChaineeTriee qui est un ensemble correspondant à
// l'union de cet ensemble et de ensembleB
et la tester avec testeUnionAvec() de exercice 3.
```

VIII. Intersection de deux ensembles triés

Ajouter, en l'implantant, la méthode suivante :

```
ListeChaineeTriee<TypeInfo>*
```

```
intersectionAvec(ListeChaineeTriee<TypeInfo>* ensembleB);
// retourne une ListeChaineeTriee qui est un ensemble correspondant à
// l'intersection de cet ensemble et de ensembleB
```

et la tester avec testeIntersectionAvec() de exercice 3.

Page 4 sur 8 R3.02—TP1

Annexes

Annexe 1 : liste chaînée TypeInfo «interface» ListeInterface + estVide() : bool + getLongueur() : int + insereTete(TypeInfo& nouvelleInfo) + supprimeTete() + insereAtPositIter(int nouvellePosition, TypeInfo& nouvelleInfo) : bool + insereAtPositRec(int nouvellePosition, TypeInfo& nouvelleInfo) : bool + supprimeAtPositIter(const int position) : bool + supprimeAtPositRec(const int position) : bool + getInfoAtPositRec(int position) : TypeInfo + getInfoAtPositRec(int position) : TypeInfo + vide() Д TypeInfo ListeChainee + nbCellules : int - ptrTete TypeInfo Cellule 0..1 info : TypeInfo - ptrCellSuivante Cellule(TypeInfo& uneInfo) Cellule(TypeInfo& uneInfo, Cellule<TypeInfo>* ptrCelluleSuivante) CF figure suivante ... + setInfo(TypeInfo& uneInfo) + setSuivante(Cellule<TypeInfo>* ptrCelluleSuivante) + getInfo() : TypeInfo + getSuivante() : Cellule<TypeInfo>* + getRefSuivante() : Cellule<TypeInfo>*&

Figure 2 : Spécification UML des classes autour de la liste chaînée

R3.02—TP1 Page 5 sur 8

```
TypeInfo
                                                    ListeChainee
+ nbCellules : int
ListeChainee()
ListeChainee(ListeChainee<TypeInfo>& uneListeChainee)
+ estVide() : bool
+ getLongueur() : int
// INSERTION en tête
- insereTeteWorker(Cellule<TypeInfo>*& ptrCetteListe, TypeInfo& nouvelleInfo)
+ insereTete(TypeInfo& nouvelleInfo)
// SUPPRESSION en tête
- supprimeTeteWorker(Cellule<TypeInfo>*& ptrCetteListe)
+ supprimeTete()
// INSERTION à position itératif et récursif
+ insereAtPositIter(int nouvellePosition, TypeInfo& nouvelleInfo) : bool
- insereAtPositRecWorker(Cellule<TypeInfo>*& ptrCetteListe, int nouvellePosition, TypeInfo& nouvelleInfo)
+ insereAtPositRec(int nouvellePosition, TypeInfo& nouvelleInfo): bool
// SUPPRESSION à position itératif et récursif
+ supprimeAtPositIter(int position): bool
– supprimeAtPositRecWorker(Cellule<TypeInfo>*& ptrCetteListe, int position)
+ supprimeAtPositRec(int position) : bool
// NETTOYAGE 1 itératif et 1 récursif
+ videIter();
- videRecWorker(Cellule<TypeInfo>* ptrCetteListe)
+ videRec();
// PRÉSENCE D'UNE INFORMATION récursif
– estInfoPresentRecWorker(const Cellule<TypeInfo>* ptrCetteListe, const TypeInfo& infoCible): bool
+ estInfoPresenteRec(TypeInfo& infoCible) : bool
// ACCESSEUR et MUTTATEUR AUTONOMES à position pour info en version récursive avec leurs workers
- getInfoAtPositRecWorker(Cellule<TypeInfo>*& ptrCetteListe, int position)
+ getInfoAtPositRec(int position) : TypeInfo {exception PrecondVioleeExcep}
- setInfoAtPositRecWorker(Cellule<TypeInfo>*& ptrCetteListe, int position, TypeInfo& nouvelleInfo)
+ setInfoAtPositRec(int position, TypeInfo& nouvelleInfo) {exception PrecondVioleeExcep}
// AFFICHAGE 2 itératif (GD facile, DG avec une pile) et 2 récursifs (GD & DG)
+ afficheGDIter()
- afficheGDWorker(Cellule<TypeInfo>* ptrListe)
+ afficheGD()
+ afficheDGIter()
- afficheDGWorker(Cellule<TypeInfo>* ptrListe)
+ afficheDG()
```

Figure 3 : Spécification UML de la classe ListeChainee

Page 6 sur 8 R3.02—TP1

Annexe 2: pile

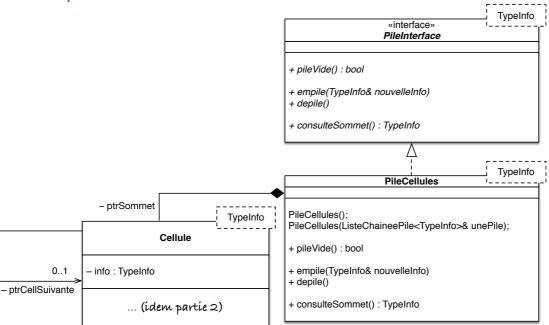


Figure 4 : Spécification UML des classes PileInterface et PileCellules

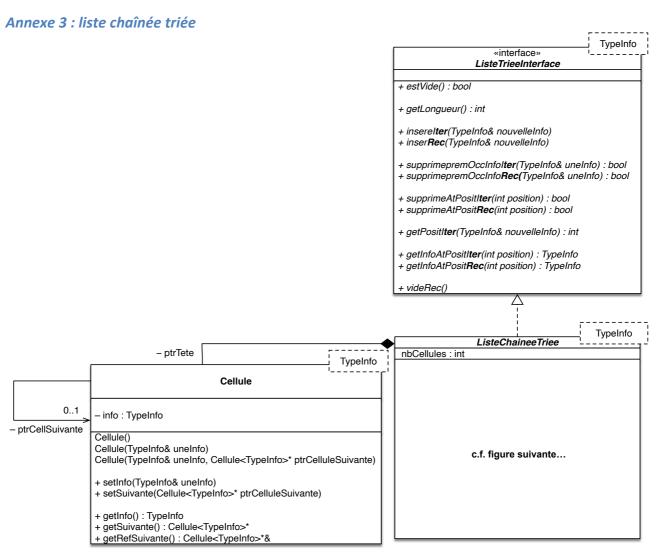


Figure 5 : Spécification UML des classes autour de la liste chaînée triée

R3.02—TP1 Page 7 sur 8

TypeInfo ListeChaineeTriee nbCellules : int ListeTrieeChainee() ListeTrieeChainee(ListeTrieeChainee<TypeInfo>& uneListe) // MÉTHODES PUBLIQUES UTILITAIRES + estVide() : bool + getLongueur() : int + vide() // ACCÈS TRIÉ ASSOCIATIF (itératif) + getPositIter(TypeInfo& uneInfo): int // INSERTION TRIÉE (récursif avec son worker) - insereRec(const TypeInfo& nouvelleInfo) - insereRecWorker(Cellule<TypeInfo>*& ptrCetteListe, TypeInfo& nouvelleInfo) // SUPPRESSION TRIÉE ASSOCIATIVE (itératif, récursif avec son worker) + supprimePremOccInfolter(TypeInfo& uneInfo): bool + supprimePremOccInfoRec(TypeInfo& uneInfo) : bool - supprimePremOccInfoRecWorker(Cellule<TypeInfo>*& ptrCetteListe, TypeInfo& uneInfo): bool // AFFICHAGE DANS L'ORDRE CROISSANT!!!!!!! + afficheCroissantRec() afficheCroissantRecWorker(Cellule<TypeInfo>* ptrCetteListe) // MÉTHODES PRIVÉES UTILITAIRES - copieListe(Cellule<TypeInfo>* listeChaineeOriginale) : Cellule<TypeInfo>* // MÉTHODES LIÉES AUX ENSEMBLES - copieListe(Cellule<TypeInfo>* listeChaineeOriginale) : Cellule<TypeInfo>* + estEnsemble(): bool + supprimeToutesDuplications(): bool + unionAvec(const ListeTrieeChainee<TypeInfo>* ensembleB): ListeTrieeChainee<TypeInfo>* + insersectionAvec(const ListeTrieeChainee<TypeInfo>* ensembleB) : ListeTrieeChainee<TypeInfo>*

Figure 6 : Spécification UML de la classe ListeChaineeTriee

Page 8 sur 8 R3.02—TP1