Marquer comme terminé

Ce livre contient les énoncés de tous les devoirs.

La classe `list`

Énoncé

Dans ce travail, vous devez compléter l'implémentation de la classe `list` fournie. La section suivante vous indique les détails de chacune des structures à implémenter.

Informations complémentaires

Vous trouverez ici le code de base qui vous permettra de respecter les directives du devoir :

Téléchargez

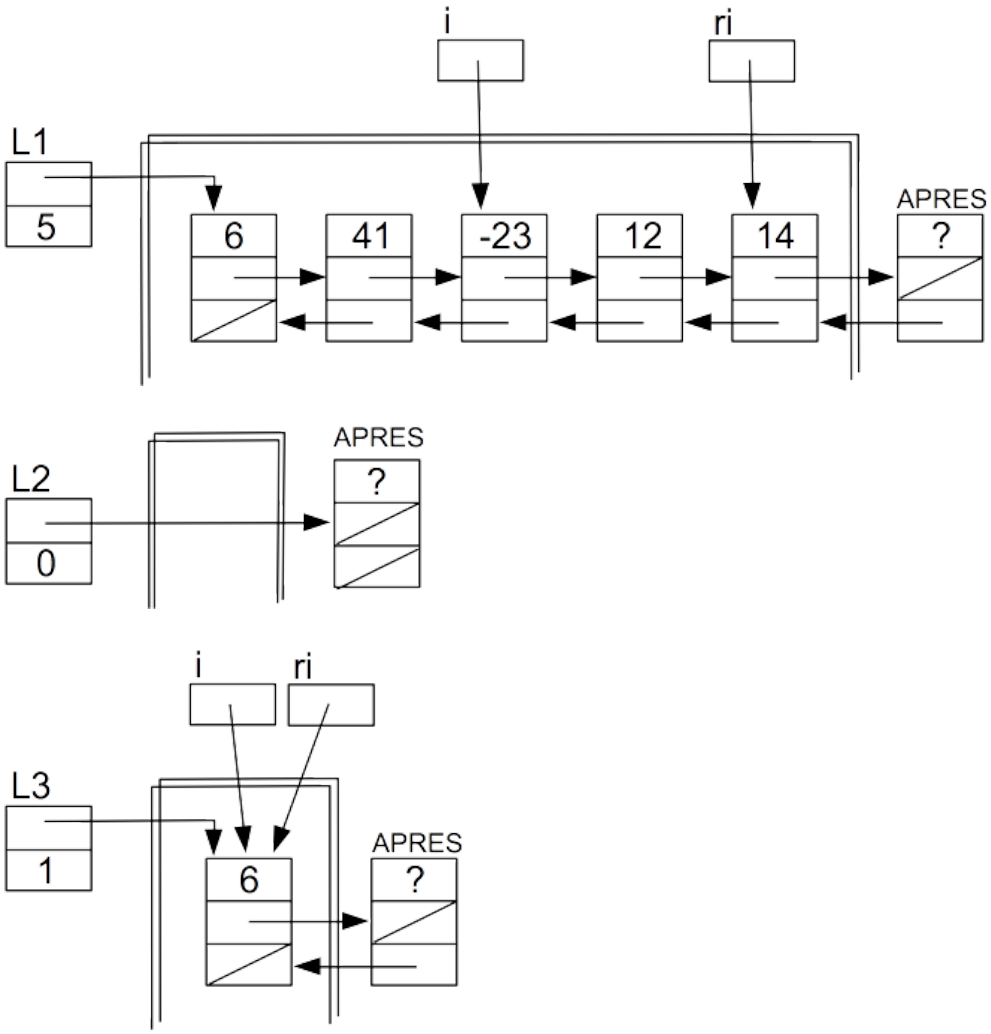
La classe `list`

Pour ce devoir, vous devez implémenter et tester une classe `list` à peu près équivalente fonctionnellement à celle de la bibliothèque normalisée (*BN*). Vous devez adopter la même représentation que ce qui est adopté en général. La liste est une structure linéaire où l'insertion et l'élimination d'un élément peuvent se faire en temps constant $\mathcal{O}(1)$. Mais la recherche par le contenu et la localisation d'un élément par son numéro prennent un temps $\mathcal{O}(n)$, car on n'a aucune façon de localiser la position d'un élément donné par un simple calcul comme pour les tableaux.

Dans la représentation classique, une cellule de queue est utilisée pour permettre d'unifier la position de la fin avec les autres positions dans la liste. Il n'y a en général pas de cellule de tête. C'est pourquoi un système de pointeurs différés est utilisé pour représenter les `reverse_iterator`. La méthode `rbegin()` donne accès au dernier élément d'une liste non vide. Sa représentation est cependant un pointeur sur la cellule suivante, la cellule de queue. Il n'est pas obligatoire de procéder de cette manière. En effet, un pointeur direct pourrait être utilisé. Cependant, le code des fonctions génératrices est plus simple à unifier en utilisant des pointeurs différés. Il y a dans la liste deux fonctions d'insertion, une avec un `iterator` et une autre avec un `reverse_iterator`. Elles n'ont pas le même comportement, mais une bonne partie de leur travail peut être codé dans une fonction de service privée qui reçoit un pointeur de cellule plutôt qu'un itérateur. La coquille publique récupère le pointeur de l'itérateur, appelle la fonction privée, puis s'occupe de retourner le bon résultat à l'usager.

La bibliothèque normalisée (*BN*) contient aussi des algorithmes sur les conteneurs, comme l'inversion de tous les éléments, leur tri, etc. La fonction `sort(i1,i2)` trie tous les éléments dans l'intervalle `[i1,i2[`. Elle ne fonctionne cependant pas avec des listes (les itérateurs de liste ne sont pas des `random_iterator`). Dans le code fourni, vous avez déjà le code d'une fonction `sort` qui prend n'importe quelle sorte d'itérateur. Vos `reverse_iterator` pourront-ils servir à faire un tri inversé avec cette fonction?

Voici quelques illustrations de représentations de listes et d'itérateurs. `L1` est une liste générale de cinq éléments, où `i` est un `iterator` qui nous donne la position de l'élément `-23` et `ri` un `reverse_iterator` qui nous donne la position de l'élément `12`. `L2` est une liste vide. `L3` est une liste d'un seul élément avec un `iterator` sur le début et un `reverse_iterator` sur la fin inversée. Pas besoin de chaine circulaire dans la représentation à cause de la cellule `APRES`.



Précisions

- Le code de base d'une classe `list` vous est fourni.
- Il vous reste à coder les fonctions génératrices privées, l'ensemble de la classe `reverse_iterator`, l'affectateur et l'algorithme `reverse`. Celui-ci inverse la liste sans bouger aucun élément, seulement les pointeurs dans les cellules.
- Le code fourni est dans deux fichiers, `list.h` et `listImpl.h`. Vous ne devez modifier QUE `listImpl.h`. Vous avez à votre disposition une fonction `afficher` qui vous permet d'aller constater si votre code a correctement modifié la liste. N'hésitez pas à vous en servir.
- Notez qu'il n'y a aucune robustesse requise dans ce code. En cas de faute, les pointeurs nuls se chargeront bien de faire planter le programme !