TD 10 - Listes

Christophe Fiorio

L'objectif est de définir et d'implémenter les types abstraits TListeEntier et ItListeEntier décrivant une liste d'entiers relatifs et un itérateur sur ce type de liste. La *spécification fonctionnelle* du type abstrait TListeEntier est la suivante :

- `créerListe` : → TListeEntier // retourne la liste vide : `estVide(créerListe())` retourne `true`.
- `estVide` : TListeEntier → bool // retourne `true` si la liste est vide, `false` sinon.
- `ajoutDébut` : TListeEntier × int → TListeEntier // retourne la liste obtenue en ajoutant l'entier donné en tête de liste.
- `ajoutFin` : TListeEntier × int → TListeEntier // retourne la liste obtenue en ajoutant l'entier donnée en fin de liste ; si la liste est vide, se comporte comme `ajoutDébut`.
- `Premier` : TListeEntier → (int|`Vide`) // retourne le premier élément de la liste ; renvoie `Vide` si la liste est vide.
- `Dernier` : TListeEntier → (int|`Vide`) // retourne le dernier élément de la liste ; renvoie `Vide` si la liste est vide.
- `nbOccurences` : TListeEntier × int → int // retourne le nombre d'occurences de l'entier donné dans la liste ; si la liste ou l'entier n'est pas présent, `nbOccurences` renvoie 0.
- `supprimerPremier` : TListeEntier → TListeEntier // retourne la liste obtenue en ayant supprimé le premier élément de la liste ; Erreur si la liste est vide.
- `supprimerDernier` : TListeEntier → TListeEntier // retourne la liste obtenue en ayant supprimé le dernier élément de la liste ; Erreur si la liste est vide.

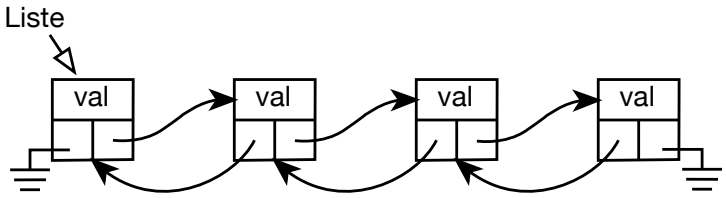

Figure 1 – liste doublement chaînée

*** Exercice 1 Description logique

Pour simplifier l'écriture des algorithmes on décide d'utiliser une liste doublement chaînée suivant le schéma de la figure 1.

On décide dans un premier temps de définir la version purement récursive du type, qu'on appellera TListeEntierRecur. On utilisera ce dernier plus tard pour définir le type concret TListeEntier.

Donnez les structures TListeEntierRecur et TListeEntierNode permettant de définir le type purement récursif, ainsi que les algorithmes associés correspondant à la spécification fonctionnelle

Vous étudierez également la complexité de toutes les fonctions et procédures écrites itérativement.

** Exercice 2 Liste circulaire avec tête de liste

Afin de donner une concrétisation à l'objet liste et afin également d'éviter les traitements particuliers liés au premier et dernier élément de la liste, on décide d'utiliser un élément fictif [1], appelé *tête de liste*, afin que la suite de la liste ne soit jamais vide physiquement. Cette tête de liste permettra de simplifier encore les algorithmes en n'ayant plus à gérer la valeur `Vide`. On pourra se servir de la valeur fictive, un entier dans notre cas, pour stocker le nombre d'éléments de la collection. La figure 2 illustre cette implémentation.

Donnez la description logique complète [2] du type TListeEntier.

1. C'est à dire un élément ne contenant aucun véritable valeur de la collection.
2. Structure de données et algorithmes.

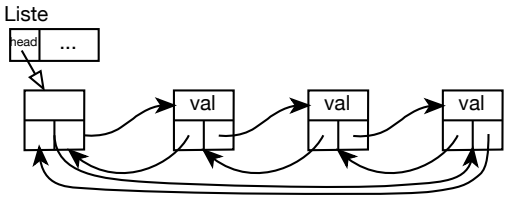

FIGURE 2 – liste circulaire doublement chaînée avec tête de liste

*** Exercice 3 Itérateur

Pour pouvoir utiliser pleinement notre type abstrait TListeEntier, nous devons pouvoir le parcourir et modifier la séquence des éléments sans connaître sa structure. Nous devons donc définir un type abstrait itérateur ItListeEntier qui fournira toutes les fonctions nécessaires à la manipulation des listes, que ce soit le parcours, ou l'insertion et la suppression d'un élément particulier.

- `créerIt` : TListeEntier → ItListeEntier // Crée un itérateur associé à la liste donnée en paramètre. Si la liste n'est pas `Vide`, l'élément courant est le premier élément de la liste.
  Si la liste est `Vide`, l'itérateur n'a pas d'élément courant :
  - `finDeParcours(créerIt(créerListe())) = True`
  - `Non(estVide(l))⇒valeur(créerIt(l)) == Premier(l)`.

Le type ItListeEntier sera quant à lui définit ainsi :

- `liste` : ItListeEntier → TListeEntier // Renvoie la liste sur laquelle travaille l'itérateur.
- `finDeParcours` : ItListeEntier → bool // Renvoie `True` si l'itérateur a terminé de parcourir la liste.
  - `finDeParcours(créerIt(créerListe())) == True`
  - `estDernier(It) Et suiv(It)⇒finDeParcours(It)`
  - `estPremier(It) Et prec(It)⇒finDeParcours(It)`
- `reinit` : ItListeEntier → ItListeEntier ×bool // Réinitialise l'itérateur sur le premier élément de la liste ; si la liste n'est pas vide, il est positionné sur le premier élément de la liste et renvoie `True`, sinon il est retourne `False`.
  - `finDeParcours(reinit(créerIt(créerListe()))) = True`
  - `Non(estVide(l))⇒finDeParcours(reinit(l)) = False`
  - `finDeParcours(reinit(créerIt(créerListe()))) = True`
- `suiv` : ItListeEntier → ItListeEntier × (int|`Vide`) // Si `Non(finDeParcours(It))`, l'itérateur renvoie l'élément courant et passe à l'élément suivant, sinon il retourne `Vide`.
- `prec` : ItListeEntier → ItListeEntier × (int|`Vide`) // Même comportement que `suiv` mais en reculant dans la liste.
- `estDernier` : ItListeEntier → bool // Renvoie `True` si l'élément courant est le dernier élément de la liste ; `False` si l'itérateur n'est pas sur le dernier élément ou si `finDeParcours(It)`.
- `estPremier` : ItListeEntier → bool // Renvoie `True` si l'élément courant est le premier élément de la liste ; `False` si l'itérateur n'est pas sur le premier élément ou si `finDeParcours(It)`.
- `valeur` : ItListeEntier → (int|`Vide`) // Si `finDeParcours(It)`, l'itérateur renvoie `Vide`, sinon renvoie la valeur de l'entier courant.
- `changerValeur` : ItListeEntier × int → int // Change la valeur de l'entier courant par l'entier donné et renvoie l'entier qui a été remplacé ; Erreur si `finDeParcours(It)`.
- `insérerApres` : ItListeEntier × int → ItListeEntier // Insère l'entier donné après l'élément courant ; si la liste était vide, insère en tête de liste. Le nouvel élément inséré devient l'élément courant, et donc l'itérateur itère. Si `finDeParcours(It)`, alors ne fait rien.
- `insérerAvant` : ItListeEntier × int → ItListeEntier // Insère l'entier donné avant l'élément courant ; si la liste était vide, insère en tête de liste. Le nouvel élément inséré devient l'élément courant. Si `finDeParcours(It)`, alors ne fait rien.
- `supprimerElt` : ItListeEntier → bool // Supprime l'élément courant ; l'itérateur repère désormais l'élément suivant ; si on était sur le dernier élément, alors désormais `finDeParcours(It)` ; si la liste était vide ou si `finDeParcours(It)`, c'est une erreur. La fonction renvoie `True` si après l'opération `Non(finDeParcours(It))`.

En vous basant sur la spécification fonctionnelle des types Liste précédemment définis, et considérant que le choix a été fait d'une liste doublement chaînée circulaire avec tête de liste pour la structure de données, donnez la description logique complète (structures de données et algorithmes) du type abstrait ItListeEntier.