

Algorithmique et structures de données
TD 2 – les listes chaînées

On considère des listes d'entiers représentées par une structure simplement chaînée.
On définit la structure *nœud* suivante :

```
Structure nœud {
    valeur : entier
    suivant : pointeur sur nœud
}
```

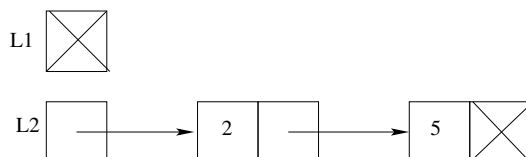
Une liste est identifiée par un pointeur sur le premier nœud.

La liste vide est le pointeur None.

On rappelle que l'on définit le type liste avec l'instruction

type liste = pointeur sur nœud

Question 1. Écrivez les instructions nécessaires pour obtenir les listes *L1* et *L2* suivantes



On définit la procédure *creerListe* suivante

1 *creerListe*(*n* : entier) : liste

nouvelleListe : liste

nouvelleListe = None

tmp : pointeur sur nœud

pour *i* de *n* à 1 par pas de -1 **faire**

 Nouveau(tmp)

 tmp→valeur = *i*

 tmp→suivant = nouvelleListe

 nouvelleListe = tmp

retourner nouvelleListe

Question 2. Détaillez l'exécution pas à pas du programme suivant

L : liste ; L = *creerListe*(3)

Question 3. Dessinez la liste *L* après la deuxième et la cinquième instruction du programme suivant :

1: L : liste

2: L = *creerListe*(5)

3: P : pointeur sur nœud

4: P = L

5: P→suivant→val = 8

Question 4. Que fait la procédure *mystere* suivante ?

Exprimez en fonction de la longueur de la liste le nombre de fois que l'on effectue l'instruction $L = L \rightarrow \text{suivant}$.

Donnez une version récursive de *mystere*.

2 *mystere*($L : \text{liste}$) : entier

compteur : entier

compteur = 0

tant que $L \neq \text{None}$ **faire**

 compteur = compteur + 1

$L = L \rightarrow \text{suivant}$

retourner compteur

Question 5. Écrivez une procédure récursive *affichePairsRec* qui prend en argument une liste et qui affiche dans le même ordre les éléments de la liste qui sont de valeur paire (les éléments de valeur impaire ne sont donc pas affichés). On définit comme coût le nombre de nœuds visités. Quel est le coût de *affiche_pairsRec* ?

Question 6. Écrivez *affichePairsIt* la même procédure mais sous forme itérative.

Question 7. Écrivez une procédure récursive *affichePairsInverseRec* qui prend en argument une liste et qui affiche dans l'ordre inversé les éléments de la liste qui sont de valeur paire (les éléments de valeur impaire ne sont pas affichés).

Question 8. * Écrivez une procédure itérative *ajoutEnFinIt* qui prend en argument une liste L et un entier x et qui retourne la liste L obtenue après insertion de x en fin de la liste. Donnez une version récursive de *ajoutEnFinIt*.

Question 9. Écrivez une procédure *testCroissant*($L : \text{liste}$) : *booléen* qui retourne VRAI si les valeurs des nœuds de la liste sont rangés par ordre croissant et FAUX sinon. Par convention, nous retournerons VRAI si la liste est vide ou constituée d'un seul nœud.

Question 10. * Écrire une procédure récursive *copiePairs*($L : \text{liste}$) : *liste* qui prend en argument une liste L et retourne une nouvelle liste formée des éléments de L de valeur paire, dans le même ordre. On ne modifiera pas la liste L .

Indication : pour cela, on distinguera les trois cas suivants :

1. la liste est vide
2. le premier élément de la liste est un entier impair
3. le premier élément de la liste est un entier pair.

Exercice 1. Liste d'entiers

Question 1. Rappelez la structure *nœud* et le type *liste* utilisés pour représenter une suite finie d'entiers par une liste simplement chaînée.

Question 2. Écrivez une procédure récursive ou itérative *copieListe(L : liste) : liste* qui prend en argument une liste *L* et retourne une copie de *L*. Attention ! la copie doit être physiquement distincte de *L*, c'est-à-dire sans aucun nœud en commun.

Question 3. Écrivez une procédure récursive *supprimeListe(L : liste, x : entier) : liste* qui supprime toutes les occurrences de la valeur *x* dans une liste *L*. Cette procédure prend en entrée une liste *L* et un entier *x* et retourne la liste *L* privée de toutes les occurrences de *x* (la liste *L* est donc modifiée).

Question 4. * Donnez une version itérative de cette procédure.

Exercice 2. Liste ordonnées d'entiers

On s'intéresse ici à des listes d'entiers triés par ordre croissant (au sens large). On utilise des listes simplement chaînées (voir exercice précédent).

On souhaite fusionner deux listes triées *L1* et *L2* en une liste *L*. Par exemple,

- *L1* contient les entiers 1, 3, 7, 7, 11
- *L2* contient les entiers 2, 3, 4, 8, 12
- *L* contiendra les entiers 1, 2, 3, 3, 4, 7, 7, 8, 11, 12.

Question 1. Proposez un algorithme pour réaliser cette fusion et exécutez-le sur l'exemple précédent. Quelle est la complexité de l'algorithme ? Précisez les opérations que vous comptez.

Question 2. Écrivez une procédure récursive *fusionListeTriees(L1, L2 : liste) : liste* qui prend en argument deux listes triées *L1* et *L2* et retourne une nouvelle liste fusion de ces deux listes.

Attention ! La liste fusionnée ne doit pas avoir de nœud en commun avec les listes *L1* et *L2*.

Indication : vous distinguerez les cas suivants :

1. au moins l'une des listes est vide.
2. aucune des deux listes n'est vide.

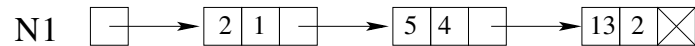
Dans le cas 1, vous pourrez utiliser la procédure *copieListe(L : liste) : liste* de l'exercice précédent.

Dans le cas 2, déterminez quel entier est ajouté à la liste fusionnée.

Exercice 3. Entiers représentés par leur décompositions en facteurs premiers

Dans cet exercice, on code des entiers naturels strictement positifs comme des produits de puissances de facteurs premiers. Plus précisément, chaque entier est représenté par une structure de liste simplement chaînée où chaque nœud contient un facteur premier de cet entier et sa puissance. On supposera que la liste est ordonnée de façon croissante suivant les facteurs premiers.

L'entier 1 est ainsi représenté par la liste vide et l'entier $N1 = 211250 = 2 * 5^4 * 13^2$ est représenté par la liste suivante où les facteurs premiers des nœuds sont rangés par ordre croissant ($2 < 5 < 7$).



Question 1. Définir la structure `nœudFacteur` adéquate.

On définit le type suivant :

`type listeFacteurs = pointeur sur nœudFacteur`

Question 2. Écrivez une procédure *valeur*($N : \text{listeFacteurs}$) : *entier* qui prend en entrée un entier codé par sa liste de facteurs et qui retourne la valeur de cet entier. Par exemple, *valeur*(N1) retournera 211250.

Question 3. Écrivez une procédure *plusGrandFacteur* ($N : \text{listeFacteurs}$) : *entier* qui retourne le plus grand facteur premier qui divise l'entier codé par N .

Question 4. Écrivez une procédure *produitEntiers*($N1, N2 : \text{listeFacteurs}$) : *listefacteurs* qui retourne $N3$, la *listeFacteurs* du produit des deux entiers codés par $N1$ et $N2$. Autrement dit, $N3$ est la liste de facteurs vérifiant

$$\text{valeur}(N1) * \text{valeur}(N2) = \text{valeur}(N3).$$

Exercice 4. Listes doublement chaînées

On définit une liste doublement chaînée à partir de la structure suivante :

```
Structure nœudDouble {  
    valeur : entier  
    precedent : pointeur sur nœudDouble  
    suivant : pointeur sur nœudDouble  
}
```

`type listeDouble = pointeur sur nœudDouble`

Question 1. Dessinez la représentation d'une liste doublement chaînée contenant les valeurs 5, 8, 3.

Question 2. * Écrivez une procédure *concateneListes*($L1, L2 : \text{listeDouble}$) : *listeDouble* qui effectue la concaténation de deux listes doublement chaînées $L1$ et $L2$.

Question 3. * Écrivez une procédure *insereListeDouble*($L : \text{listeDouble}, x : \text{entier}$) : *listeDouble* qui insère un nouvel élément x dans une liste doublement chaînée triée (les valeurs sont rangées par ordre croissant au sens large).