Corrigé du TD 4

Exercice 1.

Question 1. *Écrire une procédure qui vérifie qu'une chaîne de caractère est bien parenthésée, en utilisant un compteur.*

La variable `compteur` compte le nombre de parenthèses ouvertes qui n'ont pas encore été refermée. Il y a deux types de tests à faire : quand on ferme une parenthèse, il doit y avoir avant une parenthèse ouverte et pas encore refermée ; à la fin de la chaîne toutes les parenthèses doivent avoir été refermées.

```
bienParenthésé1(texte : chaîne de caractères) : booléen
    compteur : entier ; compteur = 0
    L : entier ; L = longueur(texte)
    i = 0
    tant que i < L faire
        si texte[i] == '(' alors
            compteur = compteur + 1
        si texte[i] == ')' alors
            si compteur > 0 alors
                compteur = compteur - 1
            sinon retourner faux
        i = i + 1
    retourner non(compteur > 0)
```

Question 2. *Peut-on procéder de même quand il y a deux type de parenthèses,* `()` *et* `{}` *?*

Un compteur ne suffit pas car il ne permet pas de distinguer les deux types de parenthèses. Même deux compteurs ne seraient pas suffisants — par exemple pour détecter une erreur sur la chaîne `{ab)`...

Question 3. *Écrire une procédure qui vérifie qu'une chaîne est bien parenthésée, avec deux types de parenthèses* `()` *et* `{}`*. On utilisera une pile.*

```
bienParenthésé2(texte : chaîne de caractères) : booléen
    P : pile ; P = initPile()
    L : entier ; L = longueur(texte)
    # on parcourt la chaîne
    i = 0
    tant que i < L faire
        # on empile les parenthèses ouvrantes
        si texte[i] == '(' ou texte[i] == '{' alors
            P = empiler(P, texte[i])
        # quand on rencontre une parenthèse fermante, on essaie de dépiler
        # la parenthèse ouvrante correspondante. Si ce n'est pas possible,
        # il y a une erreur de parenthésage.
        si texte[i] == ')' alors
            si pileVide(P) alors retourner faux
            si sommet(P) == '(' alors
                P = dépiler(P)
            sinon retourner faux
        si texte[i] == '}' alors
            si pileVide(P) alors retourner faux
            si sommet(P) == '{' alors
                P = dépiler(P)
            sinon retourner faux
        i = i + 1
    # on vérifie que toutes les parenthèses ouvrantes on été fermées
    si pileVide(P) alors retourner vrai
    sinon retourner faux
```

Exercice 2. *On considère la procédure suivante :*

```
affichageInverseRec(L : liste)
    si L <> None alors
        affichageInverseRec(L->suivant)
        afficher L->valeur
```

Question 1. *Exécuter pas à pas* `affichageInverseRec` *sur une liste à 3 nœuds. Que fait cette procédure ?*

L'exécution sur une liste `L` à 3 nœuds se déroule comme suit :
- appel de `affichageInverseRec(L)`
- appel de `affichageInverseRec(L1)`, avec `L1 = L->suivant` (liste des deux derniers nœuds de `L`)
- appel de `affichageInverseRec(L2)`, avec `L2 = L->suivant->suivant` (dernier nœud de `L`)
- appel de `affichageInverseRec(L3)`, avec `L3 = L->suivant->suivant->suivant = None`
- pour l'instant on a 4 appels « en cours » de la procédure mais on n'a encore rien affiché !
- dans le dernier appel `affichageInverseRec(L3)` le test est négatif, on ne fait rien
- on quitte `affichageInverseRec(L3)`, on revient dans `affichageInverseRec(L2)`
- on affiche `L2->valeur` qui est la dernière valeur de `L`
- on quitte `affichageInverseRec(L2)`, on revient dans `affichageInverseRec(L1)`
- on affiche `L1->valeur` qui est la deuxième valeur de `L`
- on quitte `affichageInverseRec(L1)`, on revient dans `affichageInverseRec(L)`
- on affiche `L->valeur` qui est la première valeur de `L`

On a donc affiché les valeurs de la liste `L` à l'envers. L'argument `L` de la procédure porte toujours le même nom mais renferme différentes valeurs selon le niveau de récursivité auquel on se trouve.

Question 2. *Concrétement, une pile d'appels implicite est constituée lors de l'exécution. Elle mémorise, lors de l'appel récursif, le contexte nécessaire à la reprise de la procédure au retour de cet appel. Dessiner le contenu de la pile sur l'exemple précédent.*

Lorsqu'on effectue le test dans le dernier appel `affichageInverseRec(L3)` la pile ressemble à :

`L = None`
`L = ->a`
`L = ->a->b`
`L = ->a->b->c`

À ce moment-là, la procédure ne « voit » que la valeur de `L` qui est sur le dessus de la pile, c'est-à-dire `None`. Au fur et à mesure qu'on quitte les appels récursifs, la pile est dépilée et `L` reprend les valeurs antérieures.

Question 3. *Donner une version itérative de* `affichageInverseRec` *en utilisant une pile d'entiers.*

L'idée est de faire deux boucles : une première pour parcourir la liste `L` et remplir une pile `P` avec ses valeurs, une deuxième pour dépiler `P` et afficher les valeurs au fur et à mesure qu'on les dépile. Comme on utilise une pile, les valeurs ressortent « à l'envers ».

```
affichageInverseIt(L : liste)
    P : pile ; P = initPile()
    Q : pointeur sur noeud ; Q = L
    tant que Q <> None faire
        P = empiler(P, Q->valeur)
        Q = Q->suivant
    tant que non pileVide(P) faire
        afficher sommet(P)
        P = dépiler(P)
```

Exercice 4. *Utilisation d'une liste chaînée à deux pointeurs pour représenter une file.*

Question 1. *Rappeler comment on définit une liste chaînée avec deux pointeurs* `début` *et* `fin`.

```
structure noeud
    valeur : entier
    suivant : pointeur sur noeud

structure liste2
    début : pointeur sur noeud
    fin : pointeur sur noeud
```

Question 2. *Comment fait-on pour insérer un nœud en début et en fin de liste ?*

Dans les deux cas on « fabrique » le nouveau nœud `tmp`. Toute la difficulté est de réaliser ensuite les « branchements » correctement... On notera le cas particulier de la liste vide, donnée par `L.début` = `L.fin` = `None`.

```
insereDebut(L : liste2, x : entier)
    tmp : pointeur sur noeud ; tmp = Nouveau(noeud)
    tmp->valeur = x
    tmp->suivant = L.début
    L.début = tmp
    si L.fin == None alors
        L.fin = tmp
    retourner L

insereFin(L : liste2, x : entier)
    tmp : pointeur sur noeud ; tmp = Nouveau(noeud)
    tmp->valeur = x
    tmp->suivant = None
    si L.fin <> None alors
        L.fin->suivant = tmp
        L.fin = tmp
    sinon
        L.début = tmp
        L.fin = tmp
    retourner L
```

Question 3. *Comment fait-on pour supprimer un nœud en début et en fin de liste ?*

La suppression en début de liste est très simple : si on ignore la désallocation et les cas particuliers, il suffit de faire `L.début = L.début->suivant`. La suppression en fin de liste est plus compliquée, car pour supprimer le dernier nœud on doit modifier la propriété `suivant` de l'avant-dernier nœud. Pour cela le pointeur `fin` est inutile et on doit faire un parcours de liste. On notera enfin les cas particuliers : la liste vide, et les listes à un éléments, caractérisées par l'égalité `L.début == L.fin` ($\neq$ `None`).

```
supprimeDebut(L : liste2)
    si L.début == None alors
        retourner L
    si L.début == L.fin alors
        # cas d'une liste à un élément
        désallouer(L.début)
        L.début = None
        L.fin = None
        retourner L
    # si on veut désallouer le noeud supprimé, il faut enregistrer son emplacement :
    tmp = L.début
    L.début = L.début->suivant
    désallouer tmp
    retourner L

supprimeFin(L : liste2)
    si L.début == None alors
        retourner L
    si L.début == L.fin alors
        désallouer(L.début)
        L.début = None
        L.fin = None
        retourner L
    P : pointeur sur noeud; P = L.début
    tant que P->suivant->suivant <> None faire
        P = P->suivant
    désallouer P->suivant
    P->suivant = None
    L.fin = P
    retourner L
```

Question 4. *Parmi les quatre opérations des deux questions précédentes, laquelle est coûteuse ?*

L'opération coûteuse (linéaire en la longueur de la liste) est la suppression en fin.

Question 5. *En déduire l'implémentation d'une file avec une liste chaînée à deux pointeurs : définir les cinq procédures permettant de manipuler une file.*

Comme annoncé dans le CM4, on implémente une file en temps constant. L'entrée de la file correspond à la fin de la liste, et le sommet de la file correspond au début de la liste — on entre donc dans la file par la fin de la liste, et on en sort par le début. On pourrait faire l'inverse, mais le défilage correspondrait alors à une suppression en fin de liste, ce qui est plus coûteux.

```
type file = liste2

initFile() : file
    L : file
    L.début = None
    L.fin = None
    retourner L

fileVide(L : file) : booléen
    retourner L.début == None

tête(L : file) : entier
    si fileVide(L) alors erreur('La file est vide !')
    retourner L.début->valeur

enfiler(L : file, x : entier) : file
    L = insereFin(L, x)
    retourner L

défiler(L : file) : file
    si fileVide(L) alors erreur('La file est vide !')
    L = supprimeDebut(L)
    retourner L
```