Exercice 1

Problème

Fonctions insérer en queue d'une liste simplement chaînée mais en considérant deux représentations chaînées de liste d'entiers.

V1 – La liste en tant que pointeur sur le premier élément de la liste

```
typedef struct elem {
        int value ;
        struct elem* next ;
} Element ;
typedef Element* ListOfInteger ;

Fonction insérerQueue(l : Liste<Entier>, e : Entier) : Liste<Entier>
Début
        newel : Element ← créerElement()
        valeur(newel) ← e
        succ(newel) ← INDEFINI

        Si estVide(l) alors
                l ← newel (ou tete(l) ← newel)
        sinon
                p : Element ← tete(l)
                TantQue non estVide(succ(p)) faire
                        p ← succ(p)
                fait
                succ(p) ←newel
        FinSi
        insérerQueue ← l
Fin.
```

V2 – La liste en tant que structure à deux champs (un pointeur sur le premier élément, la taille de la liste)

```
typedef struct elem {
        int value ;
        struct elem* next ;
} Element ;
```

Element = Valeur(Entier) x Suivant(Element)

```
typedef struct {
        Element* head ;
        int size ;
} ListOfInteger ;
```

ListOfInteger = Element x Taille(Entier)

```
Fonction insérerQueue(l : Liste<Entier>, e : Entier) : Liste<Entier>
Début
        newel : Element ← creerElement()
        valeur(newel) ← e
        succ(newel) ← INDEFINI

        Si estVide(l) alors
                tete(l) ← newel
                taille(l) ← 1
        sinon
                p : Element ← tete(l)
                TantQue non estVide(succ(p)) faire
                        p ← succ(p)
                fait
                succ(p) ←newel
                taille(l) ← taille(l) + 1
        FinSi
        insérerQueue ← l
Fin.
```

Exercice 2

Problème
Écrire l'algorithme du sous-programme Tri Bulle pour trier une liste d'entier.

Fonction bubbleSort(l : Liste<Entier> non triée) : Liste<Entier> triée dans l'ordre croissant
Début

```
p : Element ← tete(l)
swap : Boolean ← Faux
tmp : Entier ← 0

Si non estVide(l) ET non estVide(succ(p)) alors
    Faire
        swap ← faux
        p ← tete(l)
        TantQue non estVide(succ(p)) faire
            Si valeurTete(p) > valeurTete(succ(p)) alors
                tmp ← valeurTete(p)
                valeurTete(p) ← valeur(succ(p))
                valeur(succ(p)) ← tmp
                swap ← Vrai
            FinSi
            p ← succ(p)
        Fait
    TantQue (swap)

FinSi

bubbleSort ← l
```

Fin.

Exercice 3

Problème
Écrire l'algorithme du sous-programme Tri par Insertion pour trier une liste d'entier.

Tout d'abord l'insertion d'un nouvel entier dans une liste déjà triée d'entier.

Fonction insertIntoSortedList(l :Liste<Entier>, e : Entier) : Liste<Entier>
Début

```
newel : Element ← créerElement()
valeur(newel) ← e
succ(newel) ← INDEFINI

Si estVide(l) alors
        l ← newel
sinon
        p : Element ← tete(l)
        TantQue non estVide(succ(p)) ET valeurTete(succ(p))<value faire
                p ← succ(p)
        fait
        Si estVide(succ(p)) alors
                succ(p) ← newel
        sinon //  valeurTete(succ(p))>=value
                succ(newel) ← succ(p)
                succ(p) ← newel
        finSi

finSi

insertIntoSortedList ← l
```

Fin.

On modifie cette fonction pour l'insertion d'un nouvel élément et pas juste d'un entier.

```
Fonction insertIntoSortedList(l :Liste<Entier>, e : Element) : Liste<Entier>
Début
    Si estVide(l) alors
        l ← newel
    sinon
        p : Element ← tete(l)
        TantQue non estVide(succ(p)) ET valeurTete(succ(p))<value faire
            p ← succ(p)
        fait

        Si estVide(succ(p)) alors
            succ(p) ← newel
        sinon //  valeurTete(succ(p))>=value
            succ(newel) ← succ(p)
            succ(p) ← newel
        finSi

    finSi

    insertIntoSortedList ← l

Fin.
```

Et on utilise cette dernière fonction pour écrire le tri par insertion sur une liste, en reconstruisant une nouvelle liste avec les éléments de la liste en entrée, élément par élément.

```
Fonction insertSort(l :Liste<Entier>) : Liste<Entier>
Début
    newL : Liste<Entier> ← INDEFINI | créerListe()
    p : Element ← tete(l)

    TantQue non estVide(p) faire
        newL ←  insertIntoSortedList(newL, p)
        p ← succ(p)
    fait

    insertSort ← newL ;
Fin.
```