Algorithmique et programmation procédurale

TD No 8 : Révision - CORRIGE

Partie 1 : Liste, Pile, File

1. Écrire une fonction qui permet de supprimer un élément dans une liste doublement chaînée. Quelle est la complexité de cette fonction ?

Corrigé

```
Fonction supprimerListeDoublementChainee(teteListe : Pointeur, x : T) : Pointeur
Variables l, p : Pointeur
Début
        l ← teteListe
        Tantque (l ≠ NULL) et (valeur(l) ≠ x)
                l ← suivant(l)
        FinTantque
        Si (l = NULL) Alors
                // x n'est pas dans la liste
                retourner teteListe
        Sinon
                // x est dans la liste
                Si (precedent(l) = NULL) Alors
                        // x est le premier élément de la liste
                        teteListe ← suivant(teteListe)
                        precedent(teteListe) ← NULL
                Sinon Si (suivant(l) = NULL)
                        // x est le dernier élément de la liste
                        suivant(precedent(l)) ← NULL
                Sinon
                        // x est au milieu de la liste
                        suivant(precedent(l)) ← suivant(l)
                        precedent(suivant(l)) ← precedent(l)
                FinSi
                // dans les 3 cas, on libère le nœud et retourne teteListe
                liberer(l)
                retourner teteListe

        FinSi
Fin
```

2. Écrire une fonction fusionnant deux 2 piles triées en ordre croissant et renvoyant le résultat dans une nouvelle pile.

Corrigé

```
Fonction fusionPilesTriees(p1 : Pile, p2 : Pile) : Pile
Variables p, pres : Pile, e1, e2 : T
Début
        p ← pileVide()
        Tantque ( ! estVide(p1)) et ( ! estVide(p2))
                e1 ← sommet(p1)
                e2 ← sommet(p2)
```

```
                Si (e1 < e2) Alors
                        p ← empiler(p,e2)
                        p2 ← depiler(p2)
                Sinon
                        p ← empiler(p,e1)
                        p1 ← depiler(p1)
                FinSi
        FinTantque

        // continuer à ajouter les éléments de la pile restant
        Tantque ( ! estVide(p1))
                e1 ← sommet(p1)
                p ← empiler(p,e1)
                p1 ← depiler(p1)
        FinTantque
        Tantque ( ! estVide(p2))
                e2 ← sommet(p2)
                p ← empiler(p,e2)
                p2 ← depiler(p2)
        FinTantque

        // il faut renverser la pile résultat pour respecter l'ordre de tri
        pres ← pileVide()
        Tantque ( ! estVide(p))
                e1 ← sommet(p)
                pres ← empiler(pres,e1)
                p ← depiler(p)
        FinTantque
        Retourner pres
Fin
```

Partie 2 : Arbres

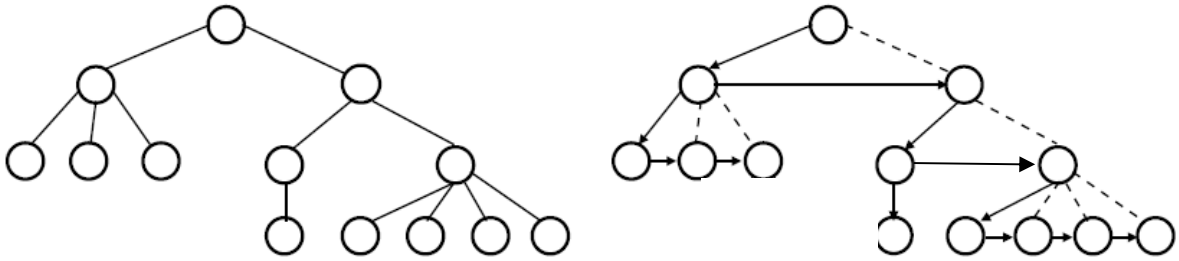
3. À partir d'un arbre binaire de recherche dont les nœuds représentent les entiers, créer un nouvel arbre binaire de recherche qui contient des entiers de signe inversé (par exemple, nœud -1 devient 1, nœud 3 devient -3, …).

Corrigé

Fonction torsion(A: **ABR**): **ABR**
Début
    Si estVide(A) **Alors**
        r**etourner** arbreVide
    Sinon
        retourner cons(-1*racine(A), torsion(fD(A)), torsion(fG(A))))
    FinSi
Fin

4. On a travaillé jusqu'à présent avec des arbres binaires dans lesquels chaque nœud avait au plus 2 sous-arbres. Nous allons maintenant généraliser la structure pour prendre en compte la possibilité pour les nœuds d'avoir un nombre quelconque de sous-arbres. On appelle de tels arbres des arbres n-aires.
   Il existe plusieurs implantations possibles d'un arbre n-aire. En voici une :

- Chaque nœud comporte un pointeur vers son premier fils et
- Chaque nœud comporte un pointeur vers son frère cadet



Arbre voulu Implémentation correspondante

On utilise la constante arbreVide et les opérations de base suivantes :

- estVide : Arbre -> Booleen
- cons : T × Arbre × Arbre -> Arbre // construire un arbre avec la racine, le premier fils et le frère cadet
- racine, r : Arbre -> T
- premierFils, pF : Arbre -> Arbre
- frereCadet, fC : Arbre -> Arbre

4.1. Écrire une fonction renvoyant le nombre de frères plus jeunes que possède un nœud donné.

4.2. Écrire une fonction renvoyant le nombre de fils que possède un nœud donné

4.3. Écrire une fonction renvoyant la hauteur d'un arbre.

Corrigé

```
Fonction nombreFreres(n : Arbre): Entier
Début
        Si  estVide(frereCadet(n))  Alors
                retourner 0
        Sinon
                retourner 1 + nombreFreres(frereCadet(n))
        FinSi
Fin

Fonction nombreFils(n : Arbre): Entier
Début
        Si  estVide(premierFils(n))  Alors
                retourner 0
        Sinon
                retourner 1 + nombreFreres(frereCadet(premierFils(n)))
        FinSi
Fin
```

```
Fonction hauteur(a : Arbre): Entier
Variables max : Entier, noeudActuel : Arbre
Début
        Si  estVide(a)  Alors
                retourner -1
        Sinon Si  estVide(premierFils(a))  Alors
                retourner 0
        Sinon
                noeudActuel  ← premierFils(a)
                max ← hauteur(noeudActuel)
                Tantque ( ! estVide(noeudActuel ))
                        noeudActuel  ← frereCadet(noeudActuel)
                        Si (hauteur(noeudActuel) > max) Alors
                                max ← hauteur(noeudActuel)

                        FinSi
                FinTantque
                Retourner 1 + max
        FinSi
Fin
```