Introduction à l'algorithmique, correction des exercices

1 Algorithmes sur les tableaux

1.1 Recherche itérative

```
entier MIN(entier t[], entier taille)
{
    entier i
    entier m = t[0]
    pour i=1 jusqu'à i<taille faire
        si t[i] < m alors
            m = t[i]
        fin si
    fin pour
    renvoyer m
}
```

1.2 Renversement d'un tableau

```
Renversement(entier T[], entier taille)
{
    entier i,tmp
    pour i=0 jusqu'à i < taille/2 faire (division entière)
        tmp = T[taille-1-i]
        T[taille-1-i] = T[i]
        T[i] = tmp
    fin pour
}
```

1.3 Palindrome

```
entier Palindrome (entier T[], entier taille)
{
    entier i,j
    i = 0
    j = taille-1
    tant que i <= j faire
        si T[i] != T[j] alors
            renvoyer 0
        sinon
            i = i+1
            j = j-1
        fin si
    fin tant que
    renvoyer 1
}
```

```
entier Palindrome (entier T[], entier taille)
{
    entier i,j
    i = 0
    j = taille-1
    tant que i <= j faire
        si T[i] = ' ' alors
            i = i+1
        sinon
            si T[j] == ' ' alors
                j = j-1
            sinon
                si T[i] != T[j] alors
                    renvoyer 0
                sinon
                    i = i+1
                    j = j-1
                fin si
            fin si
        fin si
        i = i+1
        j = j-1
    fin tant que
    renvoyer 1
}
```

1.4 Recherche dichotomique

```
entier Recherche (entier T[], entier taille, entier v)
{
    entier debut,fin,milieu
    debut = 0
    fin = taille-1
    tant que debut != fin faire
        milieu = (fin - debut) / 2
        si T[milieu] = v alors
            renvoyer milieu (la fonction s'arrête)
        fin si
        si T[milieu] < v alors
            debut = milieu
        sinon
            fin = milieu
        fin si
    fin tant que
    renvoyer -1 (la valeur n'a pas été trouvé)
}
```

– La recherche s'effectue en divisant par deux (en moyenne) le nombre d'éléments que l'on va considérer à chaque tour de la boucle "tant que". On peut donc remarquer que de le nombre d'itération est proportionnel à $log_2(n)$ (logarithme en base 2) pour un tableau de taille n.
– Cette méthode de recherche ne peut pas être appliquée sur une liste car cela nécessiterait un accès direct aux éléments de la liste (c'est à dire sans passer par le chaînage).

2 Algorithme sur les listes chaînées

2.1 Recherche dans une liste

```
entier recherche(structure liste* l, entier v)
{
    structure liste* t = l
    tant que t != NULL faire
        si (*t).valeur = v alors
            renvoyer 1
        sinon
            t = (*t).suivant
        fin si
    fin tant que
    renvoyer 0
}
```

2.2 Algorithmes sur les les piles

```
struture liste* empiler(structure liste* p, entier v)
{
    structure liste* t = AllouerNouvelElement()
    (*t).valeur = v
    (*t).suivant = p
    renvoyer t
}
```

L'appel à cette fonction se fait avec la syntaxe suivante : `l = empiler(l, valeur)` . De cette façon on ajoute en tête de liste et on maintient l'adresse de la première cellule à jour avec l'adresse renvoyée par la fonction.

```
entier sommet_pile(structure liste* l)
{
    si l != NULL alors
        renvoyer (*l).valeur
    sinon
        renvoyer erreur (on renvoie un code d'erreur)
}

structure liste* depiler(structure liste* p)
{
    structure liste* t
    si p != NULL alors
        t = p
        p = (*p).suivant
        LibererMemoire(t)
    fin si
    renvoyer p
}
```

2.3 Algorithmes sur les files

```
struture liste* ajouter_file(structure liste* f, entier v)
{
    (voir la fonction empiler)
}
```

```
entier debut_file(structure liste* f)
{
    structure liste* t = f
    si t = NULL alors
        renvoyer erreur (on renvoie un code d'erreur)
        (la fonction s'arrête ici)
    fin si
    tant que (*t).suivant != NULL faire
        t = (*t).suivant
    fin tant que
    renvoyer (*t).valeur
}

structure liste* retirer_file(structure liste* f)
{
    structure liste* t = l
    si t = NULL alors
        renvoyer erreur (on renvoie un code d'erreur)
    fin si
    si (*t).suivant = NULL alors
        LibererMemoire(t)
        renvoyer NULL
    fin si
    tant que (*(*t).suivant).suivant != NULL faire
        t = (*t).suivant
    fin tant que
    LibererMemoire((*t).suivant)
    (*t).suivant = NULL
    renvoyer f
}
```

Une liste simplement chaînée peut donc être utilisée pour avoir un comportement de file. Toutefois une amélioration notable serait de pouvoir avoir un pointeur sur la fin de la liste pour accélérer les fonctions `debut_file` et `retirer_file` .

2.4 Versions récursives des algorithmes

```
entier recherche(structure liste* l, entier v)
{
    si l = NULL alors
        renvoyer 0
    fin si
    si (*l).valeur = v alors
        renvoyer 1
    sinon
        renvoyer recherche((*l).suivant,v)
}
```

```
structure liste* inserer(structure liste* l, entier v)
{
    structure list* t
    si l = NULL ou (*l).valeur > v alors
        (si l est NULL alors (*l).valeur n'est pas évalué,
        il n'y a pas de risque d'erreur de cette façon)
        t = AllouerNouvelElement()
        (*t).valeur = v
        (*t).suivant = l
        renvoyer t
    sinon
        (*l.suivant) = inserer((*l).suivant,v)
        renvoyer l
    fin si
}
```

3 Algorithmes sur les arbres

Structure des nœuds de l'arbre :

```
structure noeud (
    entier val
    structure noeud* FilsGauche
    structure noeud* FilsDroit
)
```

3.1 Hauteur d'un arbre

```
entier hauteur(structure noeud* r)
{
    entier h
    si r = NULL alors
        h = 0
    sinon
        h = MAX(hauteur((*r).FilsGauche),hauteur((*r).FilsDroit)) + 1
    fin si
    renvoyer h
}
```

3.2 Recherche dans un arbre trié

```
entier recherche_arbre(structure noeud* a, entier v)
{
    si a = NULL alors
        renvoyer 0
    sinon si (*a).valeur = v alors
        renvoyer 1
    sinon si (*a).valeur < v alors
        renvoyer recherche_arbre((*a).FilsGauche)
    sinon
        renvoyer recherche_arbre((*a).FilsDroit)
    fin si
}
```