Master BioInformatique	ANNÉE : 2006/2007 — SESSION DE JANVIER 2007 **PARCOURS :** Master 1 **UE :** Algorithmes et structures de données **Épreuve :** Examen **Date :** Jeudi 21 décembre 2006 **Heure :** 14 heures **Durée :** 2 heures Documents : autorisés Épreuve de M. Alain GRIFFAULT

Code d'anonymat :

Avertissement

- La plus part des questions sont indépendantes.
- Le barème total est de 24 points car le sujet est assez long.
- Les questions les plus difficiles sont indiquées par un (∗).
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Exercice 1 (Listes doublement chainée (8 points))

Définition 1 *Une liste doublement chaînée est une structure dynamique* `L` *composée de cellules ayant chacune :*
- *Un champ* `info` *pour stocker les données de la liste.*
- *Un pointeur* `succ` *qui contient l'adresse de la cellule suivante dans la liste si elle existe, la valeur* `nil` *sinon.*
- *Un pointeur* `pred` *qui contient l'adresse de la cellule précédente dans la liste si elle existe, la valeur* `nil` *sinon.*

La liste `L` *est un pointeur ayant pour valeur* `nil` *si la liste est vide, sinon l'adresse de l'unique cellule dont le pointeur* `pred` *est égal à* `nil`

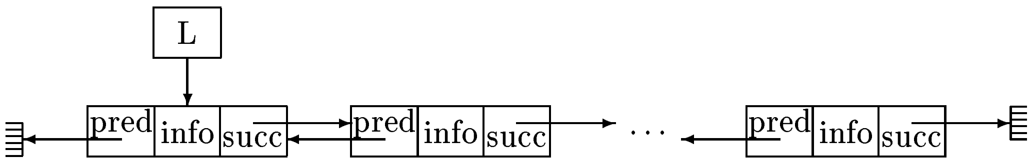

FIG. 1 – *Une liste doublement chaînée*

Les listes doublement chaînées sont manipulées par les primitives suivantes :

1. `Derniere_Cellule(L)` *qui retourne un pointeur sur la dernière cellule de la liste si elle existe, la valeur* `nil` *sinon.*
2. `Inserer_Apres(L,X,P)` *qui insère dans la liste* `L` *la cellule* `X` *après la cellule pointée par* `P`.
3. `Inserer_Tete(L,X)` *qui insère en tête de la liste* `L` *la cellule* `X`.
4. `Concatener(L1,L2)` *qui retourne le résultat de la concaténation des listes* `L1` *et* `L2` *dans la liste* `L1`.

Question 1.1 (2 points) *Écrire l'algorithme* `Derniere_Cellule(L)`.

Question 1.2 (2 points) *Écrire l'algorithme* `Inserer_Apres(L,X,P)`. *Vous supposerez que P est un pointeur valide sur une cellule de L.*

Question 1.3 (2 points) *Écrire l'algorithme* `Inserer_Tete(L,X)`.

Question 1.4 (2 points) *Écrire l'algorithme* `Concatener(L1,L2)`.

Exercice 2 (Complexité (4 points))
La fonction de Fibonacci est définie par la suite : $F_0 = 1$, $F_1 = 1$ *et* $\forall n > 1, F_n = F_{n-1} + F_{n-2}$

Question 2.1 (2 points) *Un premier programmeur propose l'algorithme suivant pour calculer le* n^{eme} *terme de la suite de Fibonacci.*

```
Fibonacci_V1(n){
    si (n > 1) alors {
        aux_2 := 1;
        aux_1 := 1;
        pour i de 2 à n faire {
            aux := aux_1 + aux_2;
            aux_2 := aux_1;
            aux_1 := aux;
        }
        retourner aux;
    }
    sinon retourner 1;
}
```

Donner et justifier la complexité (le nombre d'affectations) en fonction de `n` *de* `Fibonacci_V1`*.*

Question 2.2 (2 points (*)) *Un second programmeur propose l'algorithme suivant pour calculer le* n^{eme} *terme de la suite de Fibonacci.*

```
Fibonacci_V2(n){
    si (n > 1) alors retourner Fibonacci_V2(n-1) + Fibonacci_V2(n-2)
    sinon retourner 1;
}
```

Donner et justifier la complexité (le nombre d'appels récursifs) en fonction de F_n *de* `Fibonacci_V2`*.*

Exercice 3 (Arbres binaires (12 points))
Soit une variante de l'algorithme classique de parcours itératif d'un arbre qui retourne `vrai` *ou* `faux` *en fonction des informations contenues dans les nœuds.*

```
Parcours_Iteratif(A) {
    P <- Creer_Pile_Vide();
    Aux <- A;
    Dernier <- nil;
    tant que ((Aux <> nil) OU (non Is_Pile_Vide(P))) faire {
        tant que (Aux <> nil) faire {
            empiler(P, Aux);
            Aux <- Aux.fg;
        }
        Aux <- Sommet_Pile(P);
        depiler(P);
        si (Traitement_Noeud_Infixe(Aux, Dernier)) alors {
            Dernier <- Aux;
            Aux <- Aux.fd;
        }
        sinon
            retourner faux;
    }
    retourner vrai;
}
```

Soit l'arbre binaire suivant :

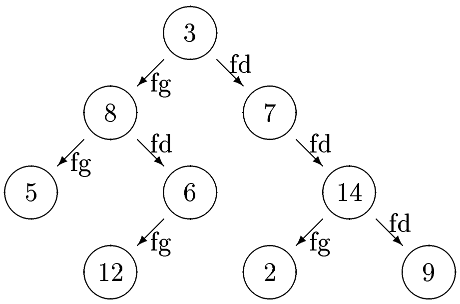

Fig. *2 – Un arbre binaire*

Question 3.1 (1 point) *Qu'affiche et que retourne l'algorithme* `Parcours_Iteratif(A)` *sur l'arbre de la figure 2, si l'on implémente* `Traitement_Noeud_Infixe(A, B)` *par :*

```
Traitement_Noeud_Affiche_Info(A, B){
    afficher A.info;
    retourner A.info < 10;
}
```

Question 3.2 (1 point) *Que représente la variable* `Dernier` *dans l'algorithme* `Parcours_Iteratif(A)`.

Définition 2 *Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tous nœud* `X`*, si* `Y` *est un nœud du sous-arbre gauche de* `X` *et* `Z` *un nœud du sous-arbre droit de* `X`*, alors* `Y.info` $\leq$ `X.info` $\leq$ `Z.info`.

Propriété 1 *Un arbre binaire est un ABR si et seulement si les informations affichées lors d'un parcours infixe sont triées.*

Question 3.3 (2 points) *Donner une implémentation* `Traitement_Noeud_ABR(A, B)`*, afin que l'algorithme* `Parcours_Iteratif(A)` *retourne vrai si et seulement si un arbre* `A` *est un ABR.*

Soit l'algorithme suivant :

```
Parcours_Avec_File(A) {
    F <- Creer_File_Vide();
    si (A <> nil) alors {
        enfiler(F, A);
        Dernier <- nil;
    }
    tant que (non Is_File_Vide(F)) faire {
        Aux <- Tete_File(F);
        defiler(F);
        si (Traitement_Noeud(Aux, Dernier)) alors {
            si (Aux.fg <> nil) alors enfiler(F, Aux.fg);
            si (Aux.fd <> nil) alors enfiler(F, Aux.fd);
        }
        sinon
            retourner faux;
        Dernier <- Aux;
    }
    retourner vrai;
}
```

Question 3.4 (1 point) *Qu'affiche et que retourne l'algorithme* `Parcours_Avec_File(A)` *sur l'arbre de la figure 2, si l'on implémente* `Traitement_Noeud(A, B)` *par :*

```
Traitement_Noeud_Affiche_Info(A, B){
    afficher A.info;
    retourner vrai;
}
```

Question 3.5 (1 point) *Que représente la variable* `Dernier` *dans l'algorithme* `Parcours_Avec_File(A)`.

Définition 3 *Un TAS est un arbre binaire qui vérifie les deux propriétés suivantes :*

P1 : pour tous nœud `X`*, si* `X.fg` *le fils gauche de* `X` *existe alors* `X.fg.info` $\leq$ `X.info` *et si* `X.fd` *le fils droit de* `X` *existe, alors* `X.fd.info` $\leq$ `X.info`*,*

P2 : l'arbre binaire est presque complet (seul le dernier niveau est éventuellement incomplet en partant des feuilles de droite).

Question 3.6 (2 points) *Donner une implémentation* `Traitement_Noeud_P1(A, B)`*, afin que l'algorithme* `Parcours_Avec_File(A)` *retourne vrai si et seulement si un arbre* `A` *vérifie la propriété P1.*

Question 3.7 (2 points (∗)***)*** *Donner une implémentation* `Traitement_Noeud_P2(A, B)`*, afin que l'algorithme* `Parcours_Avec_File(A)` *retourne vrai si et seulement si un arbre* `A` *vérifie la propriété P2.*

Propriété 2 *Un TAS est trié si et seulement si les informations affichées lors d'un parcours en largeur sont triées.*

Question 3.8 (2 points (∗)***)*** *Donner une implémentation* `Traitement_Noeud_Tas_Trie(A, B)`*, afin que l'algorithme* `Parcours_Avec_File(A)` *retourne vrai si et seulement si un arbre* `A` *est un TAS trié.*