Master BioInformatique	ANNÉE : 2009/2010 — SESSION DE JANVIER 2010 **PARCOURS :** Master 1 **UE :** Algorithmes et structures de données **Épreuve :** Examen **Date :** Mercredi 6 janvier 2010 **Heure :** 14 heures **Durée :** 2 heures Documents : autorisés Épreuve de M. Alain GRIFFAULT

SUJET + CORRIGE

Avertissement

– La plupart des questions sont indépendantes.
– Le barème total est de 24 points car le sujet est assez long.
– **(*)** indique une question difficile.
– L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Exercice 1 (Piles et Files avec des listes (9 points))

Définition 1 *Une liste 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, l'adresse* `nil` *sinon.*
La liste `L` *est un pointeur ayant pour valeur* `nil` *si la liste est vide, sinon l'adresse de la cellule située en tête de la liste.*

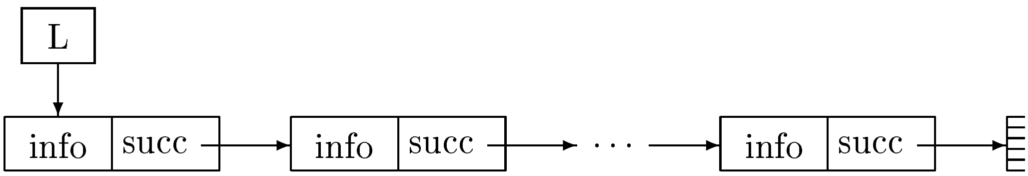

FIGURE *1 – Une liste*

Le constructeur d'une cellule s'écrit :

```
Creer_Cellule(X) {
  C.info := X;
  C.succ := nil;
  retourner C;
}
```

Une pile est une structure dans laquelle le dernier élément inséré est le premier élément retiré. Une pile peut retourner également la valeur du dernier élément inséré. Les piles sont manipulées par les primitives suivantes :

1. `Pile_Vide(P)` *qui retourne la valeur* `vrai` *si et seulement si la pile est vide.*
2. `Empiler(P,X)` *qui insère dans la pile* `P` *une cellule contenant l'information* `X`*.*
3. `Depiler(P)` *qui retire de la pile* `P` *la dernière cellule insérée si celle-ci existe.*
4. `Sommet_Pile(P)` *qui retourne l'information contenue dans la dernière cellule insérée si celle-ci existe.*

La structure de listes est très bien adaptée pour implémenter la structure de pile. Le constructeur s'écrit :

```
Creer_Pile_Vide() {
  retourner nil;
}
```

Question 1.1 (0,5 point) *Écrire la primitive* `Pile_Vide(P)`.

Réponse :

```
Pile_Vide(P) {
  retourner P=nil;
}
```

Question 1.2 (1,5 point) *Écrire la primitive* `Empiler(P,X)`.

Réponse :

```
Empiler(P,X) {
  C := Creer_Cellule(X);
  C.succ := P;
  P := C;
}
```

Question 1.3 (1 point) *Écrire la primitive* `Depiler(P)`.

Réponse :

```
Depiler(P) {
  si (Pile_Vide(P))
  alors Ecrire "Impossible : La pile est vide"
  sinon P := P.succ;
}
```

Question 1.4 (1 point) *Écrire la primitive* `Sommet_Pile(P)`.

Réponse :

```
Sommet_Pile(P) {
  si (Pile_Vide(P))
  alors Ecrire "Impossible : La pile est vide"
  sinon retourner P.info;
}
```

Une file est une structure dans laquelle l'élément retiré est le plus ancien élément inséré. Une file peut retourner également la valeur de l'élément inséré le plus ancien. Les files sont manipulées par les primitives suivantes :

1. `File_Vide(F)` *qui retourne la valeur* `vrai` *si et seulement si la file est vide.*
2. `Enfiler(F,X)` *qui insère dans la file* `F` *une cellule contenant l'information* `X`.
3. `Defiler(P)` *qui retire de la file* `F` *la cellule insérée la plus ancienne si celle-ci existe.*
4. `Tete_File(P)` *qui retourne l'information contenue dans la cellule insérée la plus ancienne si celle-ci existe.*

La structure de listes est bien adaptée pour implémenter la structure de file. Le constructeur s'écrit :

```
Creer_File_Vide() {
  // tete est un pointeur sur la cellule la plus ancienne de la file.
  // queue est un pointeur sur la cellule la plus recente de la file.
  F.tete := nil;
  F.queue := nil;
  retourner F;
}
```

Question 1.5 (1 point) *Écrire la primitive* `File_Vide(P)`.

Réponse :

```
File_Vide(F) {
  retourner F.tete = nil;
}
```

Question 1.6 (2 points) *Écrire la primitive* `Enfiler(P,X)`.

Réponse :

```
Enfiler(F, X) {
  C := Creer_Cellule(X);
  si File_Vide(F)
  alors {
    F.tete := C;
    F.queue := C;
  }
  sinon {
    F.queue.succ := C;
    F.queue := C;
  }
}
```

Question 1.7 (1 point) *Écrire la primitive* `Defiler(P)`.

Réponse :

```
Defiler(F) {
  if File_Vide(F)
  alors Ecrire "Impossible : la file est vide"
  sinon F.tete := F.tete.succ;
}
```

Question 1.8 (1 point) *Écrire la primitive* `Tete_File(P)`.

Réponse :

```
Tete_File(F) {
  if File_Vide(F)
  alors Ecrire "Impossible : la file est vide"
  sinon retourner F.tete.info;
}
```

Exercice 2 (Arbre de rangs : une variante des ABR (15 points))

Définition 2 *Un arbre de rangs (AdR) est un arbre binaire de recherche (ABR) dans lequel chaque cellule contient en plus du champ* `info`*, un entier* `taille` *qui représente la taille du sous-arbre enraciné à cette cellule.*

Dans cet exercice, une cellule sera définie par :
– *deux pointeurs* `fg` *et* `fd`*;*
– *un pointeur* `pere`*;*
– *une information entière* `info` *contenant l'information que l'on veut gérer ;*
– *un entier positif* `taille` *contenant en permanence la taille du sous-arbre ;*
et sera créée avec :

```
Creer_AdR(X) {
  A.pere := nil; A.fg := nil; A.fd := nil;
  A.info := X;
  A.taille := 1;
  retourner A;
}
```

Le champ taille est utile pour obtenir le rang d'une information donnée sans avoir à parcourir tout l'arbre de recherche. Le rang est toujours un entier strictement positif.

Propriété 1 *(Rappel) L'affichage infixe d'un arbre binaire est ordonné si et seulement si c'est un ABR.*

Propriété 2 *Soit A un arbre de rangs. Avec la convention* `C=nil => C.taille=0`*, pour tout nœud C de A, on a :* `C.taille = C.fg.taille + 1 + C.fd.taille`.

Les algorithmes `ABR_rechercher(A,X)`, `ABR_minimum(A)`, `ABR_successeur(A,X)` *s'appliquent sans aucune modification aux arbres de rangs.*
Les algorithmes d'insertion et de suppression doivent être complétés afin de maintenir à jour l'information sur les tailles des sous-arbres.

Question 2.1 (2 points) *Adapter l'algorithme* `ABR_inserer(A,X)` *vu en cours, pour obtenir un algorithme* `AdR_inserer(A,X)` *afin que les champs* `taille` *contiennent les bonnes valeurs après l'insertion de l'information* `X` *dans* `A`*.*

Réponse :

```
AdR_inserer(A, X) {
  P := Creer_AdR(X);
  si (A = nil)
  alors A := P;
  sinon {
    Aux1 := A;
    tant que (Aux1 != nil) faire {
      Aux1.taille := Aux1.taille + 1; // seule ligne a ajouter
      Aux2 := Aux1;
      si (X < Aux1.info)
      alors {Aux1 := Aux1.fg;}
      sinon {Aux1 := Aux1.fd;}
    }
    si (X < Aux2.info)
    alors {Aux2.fg := P;}
    sinon {Aux2.fd := P;}
    P.pere := Aux2;
  }
}
```

Question 2.2 (1 point) *Donner la complexité de votre algorithme* `AdR_inserer(A,X)`*.*

Réponse : *Theta(hauteur branche parcourue)*

Question 2.3 (3 points (*)) *Adapter l'algorithme* `ABR_supprimer(A,Z)` *vu en cours, pour obtenir un algorithme* `AdR_supprimer(A,Z)` *afin que les champs* `taille` *contiennent les bonnes valeurs après la suppression de la cellule pointée par* `Z`*.*

Réponse :

```
AdR_supprimer(A, Z) {
    // Z pointe sur la cellule de A qui doit etre supprimee
    // Aux pointera sur la cellule qui va remplacer Z
    si (Z.fg <> nil ET Z.fd <> nil)
    alors {
        Aux <- ABR_successeur(A, Z);


        // Aux n'a pas de fg
        // Aux est le fg de son pere qui existe
        // Deconnexion de Aux et connexion de son fd s'il existe
        Aux.pere.fg <- Aux.fd;
        si (Aux.fd <> nil)
        alors {Aux.fd.pere <- Aux.pere;}


        // Connexion de Aux à ses nouveaux fils
        Aux.fg <- Z.fg;
        Aux.fg.pere <- Aux;
        Aux.fd <- Z.fd;
        Aux.fd.pere <- Aux;


    }
    sinon {
        si (Z.fg = nil)
```

```
        alors {Aux <- Z.fd;}
        sinon {Aux <- Z.fg;}


    }
    // Connexion entre Aux et son nouveau pere
    si (Z.pere = nil)
    alors {A <- Aux;} // Z est la racine
    sinon si (Z.pere.fg = Z)
        alors {Z.pere.fg <- Aux;}
        sinon {Z.pere.fd <- Aux;}
    si (Aux <> nil)
    alors {Aux.pere <- Z.pere;} // Z n'etait pas une feuille


    // Isolation de la cellule Z pour pouvoir la detruire
    Z.pere <- nil;
    Z.fg <- nil;
    Z.fd <- nil;


}
```

Question 2.4 (1 point) *Donner la complexité de votre algorithme* `AdR_supprimer(A,Z)`*.*

Réponse : *Theta(hauteur branche parcourue)*
Les arbres de rangs permettent : d'une part de retrouver très rapidement une cellule non par son contenu, mais par sa position (son rang) dans l'arbre ; d'autre part de trouver la cellule ayant une position (un rang) donnée.

Question 2.5 (3 points) *Donner l'algorithme récursif* `AdR_cellule_rang(A,i)` *qui retourne un pointeur sur la cellule de rang* `i` *de l'arbre de rang* `A`*.*

Réponse :

```
AdR_Cellule_Rang(A, i) {
  si (A = nil || A.taille < i || i < 1)
  alors Ecrire "Impossible"
  sinon {
    // calcul du  rang de A
    si (A.fg = nil)
    alors rang := 1
    sinon rang := A.fg.taille + 1;
    // Comparaison du rang avec i
    si (rang = i)
    alors retourner A
    sinon {
      si (i < rang)
      alors retourner AdR_Cellule_Rang(A.fg, i)
      sinon retourner AdR_Cellule_Rang(A.fd, i - rang)
    }
  }
}
```

Question 2.6 (1 point) *Donner la complexité de votre algorithme* `AdR_cellule_rang(A,i)`*.*

Réponse : *Theta(hauteur branche parcourue)*

Question 2.7 (3 points (*)) *Donner l'algorithme itératif* `AdR_rang(A,Z)` *qui retourne le rang de la cellule pointée par* `Z` *dans l'arbre de rang* `A`*.*

Réponse :

```
AdR_Rang(A, Z) {
  // Z est un pointeur non nul sur une cellule de A
  // rang de Z dans Z
  si (Z.fg = nil)
  alors rang := 1
  sinon rang := Z.fg.taille + 1;
  // Aux va remonter jusqu'a la racine
  // A chaque étape si l'arbre gauche est "nouveau"
  // on ajoute sa taille et 1 pour Aux lui-meme.
  Aux := Z;
  tant que (Aux != A) faire {
    si (Aux = Aux.pere.fd)
    alors {
      si (Aux.pere.fg = nil)
      alors rang := rang + 1
      sinon rang := rang + 1 + Aux.pere.fg.taille
    }
    Aux := Aux.pere;
  }
  retourner rang;
}
```

Question 2.8 (1 point) *Donner la complexité de votre algorithme* `AdR_rang(A,Z)`.

Réponse : *Theta(hauteur branche parcourue)*