Programmation générique des arbres binaires de recherche AVL et applications

Introduction

Ce TP porte sur l'étude d'une structure de données appelée *arbres binaires de recherche* qui sert généralement à représenter un *dictionnaire*. Un dictionnaire permet de stocker des couples associant une donnée à une clé. Il offre la possibilité de rechercher rapidement la donnée associée à une clé (ici, on impose qu'il n'y ait qu'une donnée par clé), d'insérer un nouveau couple clé/donnée, de supprimer un couple clé/donnée à partir de la clé correspondante, etc.

Il y a plusieurs façon d'implémenter les dictionnaires. Le TPL2 du premier semestre utilisait un dictionnaire implémenté par une table de hachage. Par rapport aux tables de hachage, les implémentations basées sur des arbres de recherche offrent la possibilité d'effectuer un parcours *efficace* des couples du dictionnaire par ordre (croissant ou décroissant) sur les clés : ce parcours s'effectue en temps linéaire par rapport au nombre d'éléments du dictionnaire. Les arbres de recherche sont donc intéressants dès qu'on veut maintenir les clés triées. Toutefois, il existe des structures arborescentes appelées *trie* ou *prefix-tree* (e.g. arbre de préfixes) plus efficaces que les arbres de recherche lorsque les clés sont des mots ordonnés suivant un ordre lexicographique. Par rapport à ces structures de données, l'intérêt des arbres de recherche est d'être compatible avec un ordre (total) quelconque sur les clés.

Objectifs du TP

Pour illustrer ces propriétés des arbres de recherche, on veut ici définir un type abstrait des arbres binaires de recherche paramétré par le type des clés, par la relation d'ordre sur les clés et par le type des données. En Ada, cela nécessite d'introduire la notion de *généricité* : c'est-à-dire la possibilité de paramétrer une procédure, une fonction ou un paquetage par des constructions qui ne sont pas des valeurs du langage : types, exceptions, fonctions, procédures et paquetages. La généricité permet de réutiliser ces procédures ou paquetages dans différents contextes.

Concrètement, on va ainsi pouvoir instancier le type abstrait des arbres binaires de recherche sur des types de clés différents dans 2 applications :
- Dans une première application, la structure de dictionnaire permet d'associer un code postal à une ville et réciproquement. Ici, les clés sont des chaînes de caractères pour l'ordre lexicographique usuel.
- Dans une deuxième application, étant donné une fonction F des entiers dans les entiers, on utilise le dictionnaire pour stocker le graphe des appels à F. On utilise donc ici le dictionnaire avec des clés entières. Cette technique permet en particulier de ne pas faire de calculs redondants dans les appels récursifs de F.

Organisation du sujet

Le sujet commence par donner des compléments de cours sections 1 et 2 en illustrant les notions présentées avec du code fourni pour ce TP. Il présente ensuite section 3 les applications étudiées dans le TP. Enfin, il décrit le travail à faire section 4.

Convention sur les arbres

Dans tout le sujet, la hauteur d'un arbre désigne le nombre de nœuds sur sa plus longue branche. En particulier, la hauteur d'un arbre réduit à une feuille est 1. La hauteur de l'arbre vide est 0.

Plan du sujet

1 Introduction aux arbres binaires de recherche

1.1 Arbres binaires de recherche non équilibrés

Soit `Clef` le type des clés et soit `Donnee` le type des données. On suppose le type `Clef` muni d'une relation d'ordre totale. Concrètement, cette relation d'ordre totale est ici représentée par une fonction `Compare` qui permet en un seul appel de distinguer les trois cas de comparaisons entre deux clés :

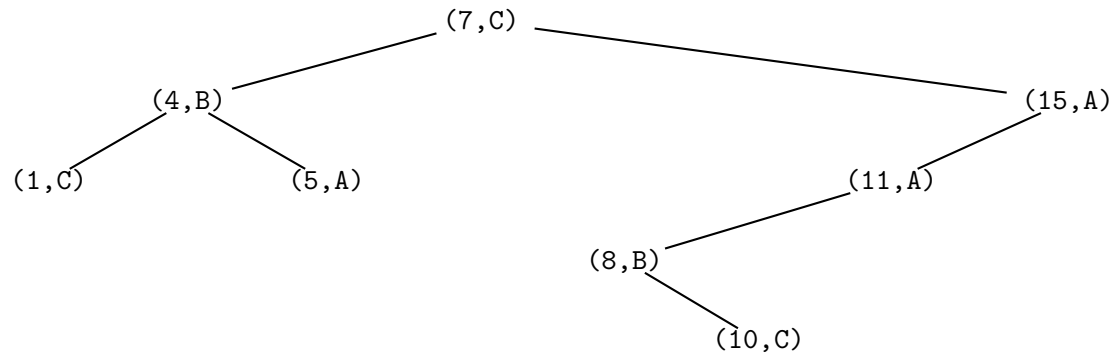
```
type Comparaison is (EQ, INF, SUP) ;
function Compare(C1,C2: Clef) return Comparaison ;
-- retourne (EQ si C1=C2) ou (INF si C1 < C2) ou (SUP si C1 > C2)
```

1.1.1 Définition

Un arbre binaire de recherche (abrégé en "ABR") est un arbre binaire A dont les étiquettes sont des couples clé/donnée et qui vérifie :
- soit A est vide
- soit le sous-arbre gauche et le sous-arbre droit de A sont deux arbres binaires de recherche, tels que toutes les clés du sous-arbre gauche sont strictement inférieures à la clé de la racine de A, et toutes celles du sous-arbre droit sont strictement supérieures à la clé de la racine de A.

En particulier, les clés sur les nœuds de A sont 2 à 2 distinctes. La figure 1 donne un exemple d'arbre binaire de recherche.

FIG. 1 – Exemple d'un arbre binaire de recherche avec clés entières



Remarquons que le parcours infixe gauche-droite sur un arbre binaire de recherche correspond au parcours des nœuds par ordre croissant sur les clés. Le parcours infixe droite-gauche correspond au parcours des nœuds par ordre décroissant sur les clés.

1.1.2 Représentation

Les traitements des ABR sont souvent symétriques sur la gauche et la droite. Pour pouvoir paramétrer les traitements par la direction (gauche ou droite), il est commode d'introduire deux constantes `Gauche` et `Droit`. Ici, ce sont des synonymes des constructeurs `INF` et `SUP` du type `Comparaison`. On définit aussi une fonction `Autre` qui retourne l'inverse d'une direction donnée.

```
subtype NomFils is Comparaison range INF..SUP ;
Gauche: constant NomFils := INF ;
Droit: constant NomFils := SUP ;
```

```
function Autre(F: NomFils) return NomFils is
   -- retourne l'autre fils que "F"
begin
   return Comparaison'Val(3-Comparaison'Pos(F)) ;
end Autre ;
pragma Inline(Autre) ;
```

Dans la suite, on représente donc les arbres binaires de recherche par le type `Dico` suivant :

```
type Noeud ;
type Dico is access Noeud ;
type TabFils is array(NomFils) of Dico ;

type Noeud is record
   C: Clef ;
   D: Donnee ;
   Fils: TabFils ;
end record ;
```

Ce type permet de représenter les arbres binaires de recherche, mais aussi bien d'autres formes d'arbres qui n'en sont pas. On donne ci-dessous une fonction qui permet de **vérifier** si un arbre `A` de type `Dico` en est un.

```
function EstABR(A:Dico) return Boolean is
begin
   if A /= null then
      Verif(A,(others => (Present => False, Valeur => A.C))) ;
   end if ;
   return True ;
exception
   when Anomalie => return False ;
end ;
```

Cette fonction utilise la procédure `Verif` définie ci-dessous. Celle-ci prend en argument `A:Dico` et `Borne: Interv`, où `Borne` représente l'intervalle sur les clefs suivant :
- soit "`Borne(INF).Valeur ... Borne(SUP).Valeur`", si "`Borne(INF).Present` **and** `Borne(SUP).Present`" ;
- soit "$-\infty$ ... `Borne(SUP).Valeur`", si "(**not** `Borne(INF).Present`) **and** `Borne(SUP).Present`" ;
- soit "`Borne(INF).Valeur` ... $+\infty$", si "`Borne(INF).Present` **and** (**not** `Borne(SUP).Present`)" ;
- soit "$-\infty \ldots +\infty$", sinon.

Ainsi, sous la précondition que `A` est un arbre non **null**, le comportement de `Verif` est le suivant :
- Soit `A` est un arbre binaire de recherche dont l'ensemble des clés se trouvent dans l'intervalle représenté par `Borne`. Dans ce cas, la procédure ne fait rien.
- Soit `A` est un arbre qui ne vérifie pas la propriété précédente. Dans ce cas, la procédure lève l'exception `Anomalie`.

```
type OptionClef is record
   Present: Boolean ;
   Valeur: Clef ;  -- a du sens uniquement si Present=True
end record ;

type Interv is array(NomFils) of OptionClef ;

Anomalie: exception ;

procedure Verif(A: Dico; Borne: Interv) is
```

```
   Aux: Interv ;
begin
   -- verification que A.C est dans les bornes
   for F in NomFils loop
      if Borne(F).Present and then Compare(Borne(F).Valeur,A.C) /= F then
         raise Anomalie ;
      end if ;
   end loop ;
   -- verification recursive sur les fils
   for F in NomFils loop
      if A.Fils(F) /= null then
         Aux(F):=Borne(F) ;
         Aux(Autre(F)):=(Present => True, Valeur => A.C) ;
         Verif(A.Fils(F), Aux) ;
      end if ;
   end loop ;
end Verif ;
```

1.1.3 Insertion d'un couple clé/donnée dans l'arbre

La procédure `Inserer` ci-dessous prend en entrée un arbre binaire de recherche `A`, une clé `C` et une donnée `D`. En sortie `A` est un arbre binaire de recherche qui a été modifié de la manière suivante :

- Si `C` figure initialement dans `A` alors l'arbre `A` est inchangé. Le paramètre de sortie `Present` prend la valeur `True`. Le paramètre `D` prend la valeur de la donnée déjà associée à `C` dans l'arbre.
- Si `C` ne figure pas dans `A`, un nouveau couple `C/D` est ajouté dans `A`. Le paramètre de sortie `Present` prend la valeur `False` et `D` n'est pas modifié.

```
1  procedure Inserer(A: in out Dico; C: Clef;
2                    Present: out Boolean; D: in out Donnee) is
3     Cmp: Comparaison ;
4  begin
5     if A = null
6     then
7        A := new Noeud'(1,C,D,(others => null)) ;
8        Present := False ;
9     else
10       Cmp:=Compare(C,A.C) ;
11       if Cmp=EQ then
12          Present:=True ;
13          D := A.D ;
14       else
15          Inserer(A.Fils(Cmp),C,Present,D) ;
16       end if ;
17    end if ;
18 end Inserer ;
```

Dans ce code, le nombre d'appels à `Compare` dans le pire cas est linéaire en fonction de la hauteur de l'arbre. Par exemple, si l'arbre est binaire complet, le coût en temps est logarithmique en fonction du nombre de nœuds. Mais si l'arbre est dégénéré, il est linéaire en fonction du nombre de nœuds.

On peut écrire une fonction de recherche d'une clé dans l'arbre en utilisant un parcours récursif similaire à celui ci-dessus. Le coût en temps dans le pire cas sera donc identique.

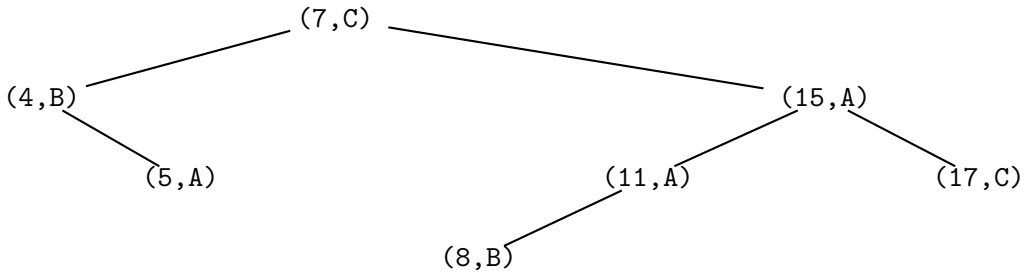
1.2 Arbres binaires de recherche équilibrés AVL

Pour garantir que les opérations de recherche et d'insertion soient logarithmiques dans le pire cas en fonction du nombre d'éléments dans l'arbre, on contraint davantage la forme des arbres : on considère ici les *arbres binaires de recherche équilibrés* introduits par Adel'son-Vel'skii et Laudis en 1962. Comme il existe maintenant d'autres variantes des arbres binaires de recherche équilibrés, on désigne sous le nom d'AVL la version originale présentée ici.

1.2.1 Définition des arbres AVL

Un arbre binaire de recherche est dit équilibré ssi lorsqu'il est non vide, ses deux sous-arbres sont équilibrés et leur différence de hauteur est d'au plus 1. L'arbre binaire de recherche de la figure 1 n'est pas équilibré, mais l'arbre de la figure 2 l'est.

FIG. 2 – Exemple d'un arbre binaire de recherche équilibré



Calculons le nombre minimal de nœuds $N(h)$ que peut comporter un arbre binaire de recherche équilibré en fonction de la hauteur h. La figure 2 donne un exemple d'un arbre équilibré avec un nombre minimal de nœuds pour une hauteur 4 : si on essaye d'enlever une feuille tout en conservant une hauteur 4, l'arbre obtenu n'est plus équilibré. On voit que $N(4) = 7$.

Plus généralement, $N(h)$ satisfait la formule de récurrence suivante :
- $N(0) = 0$ et $N(1) = 1$.
- pour $h \geq 0$, on a : $N(h+2) = 1 + N(h+1) + N(h)$. En effet, le nombre de nœuds est minimal, quand les deux sous-arbres ont un nombre minimal de nœuds est que la différence de hauteur des sous-arbres est maximale.

On vérifie facilement par récurrence que $N(h) \geq (3/2)^h - 1$.

Ceci montre que la hauteur d'un arbre binaire de recherche équilibré est logarithmique en fonction du nombre de nœuds. L'opération de recherche est donc logarithmique (c'est la même que pour les arbres de recherche non équilibrés). Il reste à voir comment insérer un couple clé/donnée dans un arbre équilibré en maintenant cet arbre équilibré (et en préservant le coût logarithmique en temps).

1.2.2 Représentation

Pour éviter d'avoir à parcourir tout un arbre pour calculer sa hauteur, on ajoute un champ `Hauteur` dans les nœuds.

```
type Noeud is record
   Hauteur: Positive ;
```

```
   C: Clef ;
   D: Donnee ;
   Fils: TabFils ;
end record ;
```

Dans les fonctions et procédures qui suivent, les données du type `Arbre` devront être des arbres binaires de recherche. On indiquera dans la spécification des procédures et fonctions si les arbres en entrée ou en sortie sont considérés comme des AVL. On respecte la convention suivante : dans tout arbre considéré comme un AVL, l'attribut `Hauteur` correspond effectivement à la hauteur de l'arbre (et ceci récursivement bien sûr).

On introduit ici quelques définitions utiles pour la suite :
– Une fonction qui calcule la hauteur d'un arbre binaire de recherche AVL :

```
function Hauteur(A: Dico) return Natural is
   -- requiert: A AVL
begin
   if A = null
   then
      return 0 ;
   else
      return A.Hauteur ;
   end if;
end ;
```

– Une procédure `AjusteHauteur` qui sert à corriger l'attribut `Hauteur` d'un arbre, quand ses sous-arbres ont été modifiés.

```
procedure AjusteHauteur(A: Dico) is
   -- requiert: A /= null et  A.Fils(Gauche), A.Fils(Droit) AVLs
   -- garantit:
   --   Si abs(Hauteur(A.Fils(Gauche),A.Fils(Droit))) <= 1 alors A AVL.
begin
   A.Hauteur:=1+Integer'Max(Hauteur(A.Fils(Gauche)),
                            Hauteur(A.Fils(Droit)));
end ;
```

1.2.3 Insertion

Si on examine la procédure d'insertion sur les arbres binaires de recherche non équilibrés section 1.1.3, on voit que les sources de déséquilibres viennent de l'appel récursif à `Inserer` ligne 15. Remarquons que si l'arbre est équilibré avant l'appel, après l'appel, comme on insère un couple dans le fils de nom `Cmp`, on a :
`Hauteur(A.Fils(Autre(Cmp))) - 1 <= Hauteur(A.Fils(Cmp)) <= Hauteur(A.Fils(Autre(Cmp))) + 2`
Il y a donc déséquilibre si `Hauteur(A.Fils(Cmp))=Hauteur(A.Fils(Autre(Cmp)))+2`. Dans ce cas, on dit que le déséquilibre est à gauche si `Cmp=Gauche`. Sinon on dit qu'il est à droite. On va écrire une procédure `Equilibrer` chargée de rééquilibrer l'arbre et paramétrée par le `Sens` du déséquilibre :

```
procedure Equilibrer(Sens: NomFils; A: in out Dico) is
-- requiert:
--       A non null et A.Fils(Gauche), A.Fils(Droit) AVLs
--       -1 <= Hauteur(A.Fils(Sens)) - Hauteur(A.Fils(Autre(Sens))) <= 2
-- garantit: A AVL
```

Sous l'hypothèse de cette procédure, la procédure d'insertion dans les AVLs s'écrit donc en ajoutant les lignes 16 à 18 ci-dessous par rapport à la version non équilibrée :

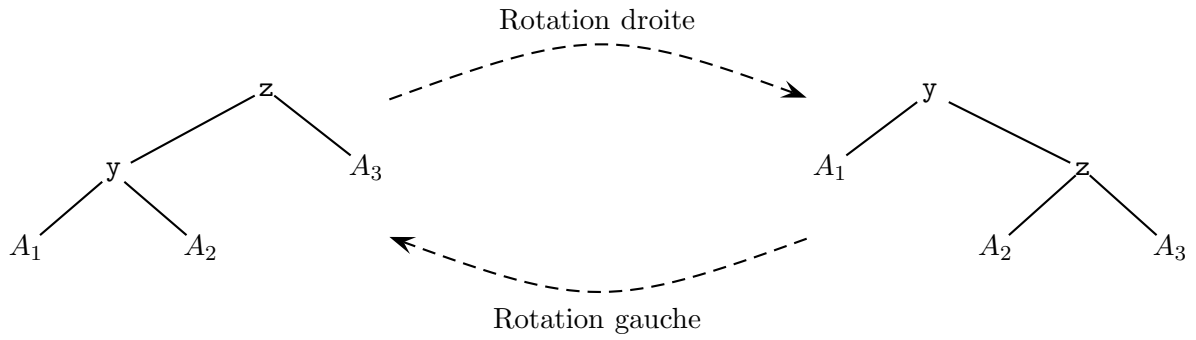
```
1  procedure Inserer(A: in out Dico; C: Clef;
2                    Present: out Boolean; D: in out Donnee) is
3     Cmp: Comparaison ;
4  begin
5     if A = null
6     then
7        A := new Noeud'(1,C,D,(others => null)) ;
8        Present := False ;
9     else
10       Cmp:=Compare(C,A.C) ;
11       if Cmp=EQ then
12          Present:=True ;
13          D := A.D ;
14       else
15          Inserer(A.Fils(Cmp),C,Present,D) ;
16          if not Present then
17             Equilibrer(Cmp,A) ;
18          end if ;
19       end if ;
20    end if ;
21 end Inserer ;
```

1.2.4 Rotations

Pour préserver le coût en temps (dans le pire cas) logarithmique de la procédure `Inserer`, la procédure `Equilibrer` doit effectuer la transformation de l'arbre à coût constant (e.g. borné par un nombre d'affectations et de comparaisons indépendant du nombre de nœuds dans l'arbre). Ces transformations ont pour briques de bases les rotations d'arbres binaires de recherche.

La figure 3 présente les 2 rotations sur les arbres binaires : rotation droite et rotation gauche. Ces deux transformations sont inverses l'une de l'autre. Sur la figure, les symboles `y` et `z` représentent 2 couples clé/donnée, et les symboles A_1, A_2 et A_3 représentent des sous-arbres. L'intérêt de ces transformations est qu'elles préservent la structure d'arbre binaire de recherche. Par ailleurs, la transformation rotation droite (respectivement gauche) augmente d'au moins 1 la taille du sous-arbre droit (resp. gauche) en diminuant d'au moins 1 la taille du sous-arbre gauche (resp. droit).

FIG. 3 – Rotations sur les arbres binaires de recherche



La rotation droite d'un arbre `A` peut s'écrire comme ci-dessous (et la rotation gauche s'écrit de manière symétrique) :

```
declare
   Fg: constant Dico := A.Fils(Gauche) ;
begin
   A.Fils(Gauche) := Fg.Fils(Droit) ;
   AjusteHauteur(A) ;
   Fg.Fils(Droit) := A ;
   AjusteHauteur(Fg) ;
   A:=Fg ;
end ;
```

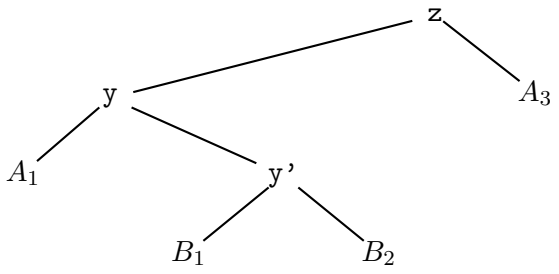
1.2.5 Équilibrage à gauche

Étudions maintenant l'équilibrage à gauche, sachant que l'équilibrage à droite va fonctionner de manière symétrique.

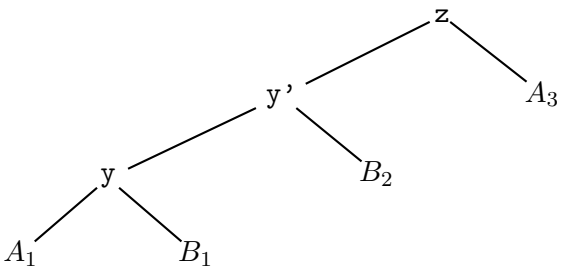
Soit `A` un arbre binaire de recherche non vide qui vient d'être déséquilibré à gauche suite à une insertion dans son sous-arbre gauche. Autrement dit :
– les arbres `A.Fils(Gauche)` et `A.Fils(Droit)` équilibrés
– et `Hauteur(A.Fils(Gauche))=Hauteur(A.Fils(Droit))+2`
Appelons `Fg` le sous-arbre gauche de `A`. La condition précédente implique :
– `Max(Hauteur(Fg.Fils(Gauche)),Hauteur(Fg.Fils(Droit)))=Hauteur(A.Fils(Droit))+1`
– et **abs**`(Hauteur(Fg.Fils(Gauche))-Hauteur(Fg.Fils(Droit))) <= 1`
Examinons à quelle condition en appliquant une rotation droite sur `A` on le rééquilibre. Ici la figure 3 correspond au cas où A_1 vaut `Fg.Fils(Gauche)`, A_2 vaut `Fg.Fils(Droit)` et A_3 vaut `A.Fils(Droit)`. On voit que la condition `Hauteur(Fg.Fils(Gauche))>=Hauteur(Fg.Fils(Droit))` est nécessaire et suffisante pour garantir que l'arbre obtenu après rotation droite de `A` est équilibré.

Supposons maintenant que `Hauteur(Fg.Fils(Gauche))<Hauteur(Fg.Fils(Droit))`. Autrement dit, on a en fait `Hauteur(Fg.Fils(Gauche))=Hauteur(A.Fils(Droit))=Hauteur(Fg.Fils(Droit))-1`.

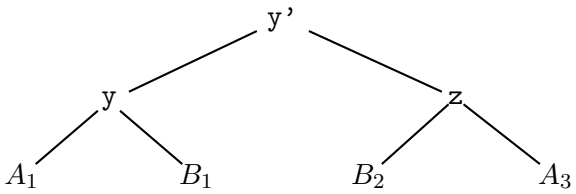
Ainsi, `A` a la forme ci-contre, où :
– les hauteurs de A_1 et A_3 sont égales ; appelons h cette donnée.
– et les hauteurs de B_1 et B_2 sont comprises entre h et $h-1$ (l'une d'elle valant nécessairement h).



Si on applique une rotation gauche sur `A.Fils(Gauche)`, l'arbre `A` est transformé en l'arbre dessiné ci-contre.



En appliquant une rotation droite sur `A`, on obtient finalement l'arbre correspondant au dessin ci-contre. Cet arbre est un arbre binaire de recherche équilibré.



L'équilibrage à gauche sur un arbre `A` s'écrit donc tout simplement en utilisant au plus 2 rotations. Son coût en temps dans le pire cas est borné par une constante.

```
declare
   Fg: constant Dico := A.Fils(Gauche);
begin
   if Hauteur(Fg) > Hauteur(A.Fils(Droit)) + 1
   then
      if Hauteur(Fg.Fils(Droit)) > Hauteur(Fg.Fils(Gauche))
      then
         -- rotation gauche sur A.Fils(Gauche)
         ...
      end if ;
      -- rotation droite sur A
      ...
   else
      AjusteHauteur(A) ;
   end if ;
end ;
```

2 Introduction à la généricité en Ada

On utilise la *généricité* quand on veut paramétrer une procédure, une fonction par des constructions qui ne sont pas des valeurs du langage (types, exceptions, fonctions, procédures et paquetages). Ça s'utilise aussi pour paramétrer un paquetage tout entier.

2.1 Paquetage générique `Dictionnaire`

Pour créer un paquetage générique, on doit déclarer les paramètres génériques sur *l'interface* du paquetage. On illustre cela sur le paquetage `Dictionnaire` implémentant les arbres binaires de recherche de la section 1. L'interface de ce paquetage, donnée ci-dessous, utilise un autre paquetage appelée `Comparaisons` qui définit notamment le type `Comparaison` (utilisé dans cette interface).

La liste des paramètres génériques est donnée sous forme d'une suite de déclarations qui commence au mot-clef **generic** et se termine ici à la première occurrence de **package** non immédiatement précédé d'un **with**. Ce mot-clef **with** sert justement à indiquer que la déclaration de procédure, de fonction ou de paquetage qui suit immédiatement est un paramètre (et pas l'unité générique).

Ici, il y a donc 3 paramètres :

1. Le type `Clef` qui du point de vue de `Dictionnaire` représente un type dont on ne connaît pas la représentation.
2. Une fonction nommée ici `Compare`.
3. Le type `Donnee` dont on ne connaît pas la représentation dans `Dictionnaire`.

```
with Comparaisons ;
use Comparaisons ;

generic

   type Clef is private ;
   with function Compare(C1,C2: Clef) return Comparaison ;

   type Donnee is private ;

package Dictionnaire is

   type Dico is private ;

   procedure Inserer(A: in out Dico; C: Clef;
                     Present: out Boolean; D: in out Donnee) ;

   -- ici, on ne detaille pas tous les sous-programmes du paquetage.

private

   -- partie privee de l'interface: on doit definir le type Dico.

   type Noeud ;

   type Dico is access Noeud ;

end Dictionnaire;
```

Dans l'implémentation du paquetage, on peut utiliser directement les paramètres génériques : ils sont implicitement dans le contexte.

```
package body Dictionnaire is

   -- ici: on commence par definir le type Noeud (introduit dans l'interface).
   subtype NomFils is Comparaison range INF..SUP ;
   type TabFils is array(NomFils) of Dico ;

   type Noeud is record
      Hauteur: Positive ;
      C: Clef ;
      D: Donnee ;
      Fils: TabFils ;
   end record ;

   procedure Inserer(A: in out Dico; C: Clef;
                     Present: out Boolean; D: in out Donnee) is
   begin
     -- cf. code dans section precedente
   end ;

end ;
```

Étant donné un tel paquetage générique `Dictionnaire`, on ne peut pas utiliser ses définitions (`Dico` et `Inserer`) directement. Il faut commencer par instancier tous les paramètres génériques.

Supposons que dans un programme `Essai1`, on veuille utiliser ce paquetage dictionnaire en instanciant `Clef` par le type `Natural`, `Compare` par une fonction `Foo`, et `Donnee` par le type `String(1..3)`. En Ada, il faut donner un nom à l'instance d'un tel paquetage générique : on choisit ici le nom `MesDicos`.

```
1  with Comparaisons, Dictionnaire ;
2  use Comparaisons ;
3
4  procedure Essai1 is
5
6     function Foo(C1,C2: Natural) return Comparaison is
7     begin
8        -- exemple trivial.
9        return EQ ;
10    end ;
11
12    subtype MesDonnees is String(1..3) ;
13
14    package MesDicos is new Dictionnaire(Natural, Foo, MesDonnees) ;
15    -- use MesDicos ;
16
17    D: MesDicos.Dico ;
18    P: Boolean ;
19    S: String := "abc" ;
20 begin
21    MesDicos.Inserer(D,10,P,S) ;
22 end ;
```

L'instanciation de `Dictionnaire` ligne 14 crée un paquetage non générique `MesDicos` qui a l'interface donnée ci-dessous. On peut donc utiliser `MesDicos` comme un paquetage usuel. En particulier, si on veut éviter d'avoir à écrire le préfixe "`MesDicos.`", il suffit de décommenter la ligne 15 ci-dessus.

```
package MesDicos is
```

```
   type Dico is private ;

   procedure Inserer(A: in out Dico; C: Natural;
                     Present: out Boolean; D: in out MesDonnees) ;
```

Remarquons qu'on doit instancier les paramètres génériques de type par des sous-types admissibles comme type de paramètre dans les sous-programmes : par exemple, il est interdit d'instancier `Donnee` par `String(1..3)` car il est interdit de l'utiliser comme type d'un paramètre d'une procédure (il faut utiliser un sous-type nommé ici). Par ailleurs, pour les paramètres de type qui sont simplement déclarés comme "**private**", l'instance du paramètre générique doit être un sous-type admissible dans les déclarations de variables : on ne peut donc pas utiliser un type tableau non-contraint comme `String` (on a par contre le droit d'utiliser un **access** `String`).

2.2 Procédures génériques et parcours séquentiels

De manière similaire aux paquetages, il est possible de définir des procédures génériques. On les présente ici sur un exemple très classique : on utilise une procédure paramétrée par une autre procédure pour faire un parcours séquentiel générique, alternativement à la définition d'une machine séquentielle.

Par exemple, considérons le fichier `codes_postaux.txt` fourni. Il contient environ 39000 lignes qui respectent la syntaxe suivante : chaque ligne commence par 5 chiffres (qui représentent un code postal), suivis par un espace, et suivi d'une suite non vide de caractères (qui représente le nom d'une ville).

```
01090 FRANCHELEINS
01000 BOURG EN BRESSE
...
```

Une première approche pour implémenter le parcours séquentiel d'un fichier respectant cette syntaxe est de définir une machine séquentielle (cf. premier semestre) dont l'interface serait donnée par :

```
package ParcourirCodesPostaux is

   procedure Demarrer(NomFichier: String) ;
   -- initialisation de la machine pour le fichier de nom NomFichier.

   function Fin return Boolean ;
   -- retourne False ssi il y a encore un code postal à lire.

   procedure Avancer ;
   -- requiert not Fin
   -- passe a la ligne suivante

   function CodePostalCourant return String ;
   -- retourne le code postal courant

   function NomVilleCourant return String ;
   -- retourne le nom de ville courant.

end ParcourirCodesPostaux ;
```

Alternativement, on peut donner écrire une procédure générique pour faire ce parcours. La procédure est paramétrée par la procédure `Traiter` qui traite chacune des lignes.

```
generic
   with procedure Traiter(CodePostal, NomVille: String) ;
procedure ParcourirCodesPostaux(NomFichier: String) ;
```

L'implémentation de la procédure est la suivante. Elle appelle séquentiellement `Traiter` sur chacune des lignes correctes du fichier. Elle lève l'exception `Constraint_Error` sur la première ligne qui ne respecte pas la syntaxe voulue.

```
with Ada.Text_Io ;
use Ada.Text_Io ;

procedure ParcourirCodesPostaux(NomFichier: String) is
   Fichier: File_Type ;
   CodePostal: String(1..5) ;
   Cour: Character ;
   Index: Natural ;
begin
   Open(Fichier, In_File, NomFichier) ;
   while not End_Of_File(Fichier) loop
      -- invariant: on est au debut d'une ligne.
      Index := 0 ;
      while not End_Of_File(Fichier) loop
         -- invariant:
         --   on lit le code postal dans la ligne courante
         --   Index: nombre de chiffres deja lus.
         Get_Immediate(Fichier,Cour) ;
         if Cour in '0'..'9' then
            Index:=Index+1 ;
            CodePostal(Index) := Cour ;
         elsif Cour = ' ' and then Index = CodePostal'Last then
            exit ;
         else
            raise Constraint_Error ;
         end if ;
      end loop ;
      -- il reste le nom de ville a lire dans la ligne
      declare
         NomVille: String := Get_Line(Fichier) ;
      begin
         if NomVille'Length <= 0 then
            raise Constraint_Error ;
         else
            Traiter(CodePostal,NomVille) ;
         end if ;
      end ;
   end loop ;
   Close(Fichier) ;
end ;
```

On peut utiliser cette procédure pour écrire un programme qui vérifie que le fichier d'entrée respecte la syntaxe, en affichant le numéro de la première ligne qui contient une erreur. On instancie ici le paramètre `Traiter` générique par une procédure appelée `InsererNouveau` qui se contente de compter combien de fois elle est appelée (variable `NbLignes`). On nomme `Analyser` la procédure qui instancie `ParcourirCodesPostaux` : le fichier appelé `NomDuFichier` est incorrect si `Analyser(NomDuFichier)` lève l'exception `Constraint_Error`. Lorsque cette exception est levée, la variable `NbLignes` indique le numéro

de la première ligne qui contient une erreur.

```
with Ada.Text_Io, ParcourirCodesPostaux ;
use Ada.Text_Io ;

procedure TestP is

   NomDuFichier: constant String := "codes_postaux.txt" ;
   NbLignes : Natural := 1 ;

   procedure InsererNouveau(C,N: String) is
   begin
      NbLignes:= NbLignes + 1 ;
      Put_Line(C & " " & N) ;
   end ;

   procedure Analyser is new ParcourirCodesPostaux(InsererNouveau) ;

begin
   Analyser(NomDuFichier) ;
exception
   when Constraint_Error =>
      New_Line ;
      Put_Line(NomDuFichier
               & ": erreur sur la ligne" & Integer'Image(NbLignes)) ;
      raise ;
end ;
```

Généralement, il est plus simple d'implémenter la procédure générique que la machine séquentielle. En particulier, la procédure générique peut être récursive. Pour implémenter la machine séquentielle correspondante, il faut donc dérécursiver le parcours en introduisant une pile explicite. Ce serait typiquement le cas pour le parcours séquentiel de l'ensemble des couples clés/données d'un arbre binaire de recherche. Définir un tel parcours par une procédure générique est très facile. Dans l'interface de `Dictionnaire`, on déclare :

```
generic
   with procedure Traiter(C: Clef; D: in out Donnee) ;
procedure Parcourir(A: Dico) ;
```

Dans l'implémentation de `Dictionnaire`, on écrit un bête parcours en profondeur :

```
procedure Parcourir(A: Dico) is
begin
   if A /= null
   then
      Parcourir(A.Fils(Gauche)) ;
      Traiter(A.C,A.D) ;
      Parcourir(A.Fils(Droit)) ;
   end if ;
end ;
```

Par contre, généralement, il est plus simple d'utiliser une machine séquentielle que d'instancier une procédure générique. En particulier, pour arrêter prématurément un parcours séquentiel effectué par une procédure générique, on est obligé de lever une exception dans la procédure qui instancie le traitement et on doit éventuellement rattraper cette exception à l'extérieur du parcours.

3 Deux applications du paquetage `Dictionnaire`

Ces deux applications permettent d'illustrer différentes instanciations du paquetage `Dictionnaire` : dans la première les clés sont des chaînes de caractères, et dans la seconde les clés sont des entiers.

3.1 Base de données des codes postaux

Il s'agit d'écrire un programme qui permet de retrouver les codes postaux associés à une commune et réciproquement les communes associées à un code postal (en effet, curieusement, il peut y en avoir plusieurs). En fait, pour faciliter l'expression de sa requête (par exemple la recherche d'un code postal), l'utilisateur peut se contenter d'écrire un préfixe (par exemple, un préfixe du nom de la commune) : le programme lui affiche tous les résultats possibles pour ce préfixe.

Concrètement, l'utilisateur voit le menu suivant :

```
---- Menu ----
0. quitter
1. recherche de code postal
2. recherche de nom de ville
--- Votre choix:
```

S'il tape "`1`", il doit ensuite saisir un préfixe de ville. S'il tape alors "`GRENO`", il voit afficher le résultat suivant :

```
+++ Codes Postaux de GRENOBLE:
38000
38100
+++ Codes Postaux de GRENOIS:
58420
```

Pour la mise en œuvre, on vous fournit un programme `codes_postaux.txt` qui contient les codes postaux des environ 36000 communes françaises. On vous fournit aussi une procédure générique de parcours de ce fichier appelée `ParcourirCodesPostaux` (voir section 2.2). Pour déboguer votre programme, il est conseillé de travailler avec un petit sous-ensemble du fichier.

Techniquement, une ville pouvant avoir plusieurs codes postaux (et réciproquement), il faut pouvoir instancier le dictionnaire avec des clés qui représentent des noms des villes et des données qui représentent des ensembles de codes postaux. Pour représenter ces ensembles, les enseignants fournissent un type abstrait permettant de représenter des ensembles de chaînes de caractères. Ce type abstrait `Ensemble`, défini dans le paquetage `EnsString`, est lui-même construit comme une instance du dictionnaire. Par ailleurs, comme expliqué en section 2.1, le compilateur interdit d'instancier le paramètre `Clef` du paquetage `Dictionnaire` par le type `String`. A la place, on peut utiliser le type `Element` du paquetage `EnsString` qui est défini comme **access** `String`. Le paquetage `EnsString` fournit en outre une fonction `Centralise` qui permet d'associer à chaque valeur de type `String` un pointeur unique de type `Element`. Utiliser cette fonction permet donc de ne pas dupliquer les chaînes de caractères dans le tas.

3.2 Mémoïzation

La mémoïzation est un procédé qui consiste à enregistrer dans un dictionnaire le graphe de tous les appels d'une fonction. Ainsi, au lieu de refaire des calculs déjà effectués, on se contente de retourner les

résultats précédemment enregistrés. Cette technique permet en particulier d'éviter automatiquement les calculs redondants dans les calculs récursifs.

Concrètement, il s'agit ici de réaliser l'implémentation du paquetage générique `Memoize` suivant :

```
generic

   with function F(N: Natural) return Natural ;

package Memoize is

   function Eval(N: Natural) return Natural ;
   -- retourne F(N)
   -- utilisez "Eval(N)" au lieu de "F(N)" pour beneficier de la memoization.

   procedure GrapheAppels ;
   -- affiche la liste des (N,F(N))
   -- pour tous les N pour lesquels il y a eu appel a "Eval(N)".

   procedure GrapheAppels(A,B:Natural) ;
   -- meme chose que ci-dessus mais uniquement pour les N de l'interv. [A,B].

end ;
```

Pour l'instancier sur la fonction de Fibonacci, il suffit de créer une instance nommée `Fib` du paquetage `Memoize` où `F` est instanciée par une fonction `DeplieFib`. Cette fonction `DeplieFib` est en fait mutuellement récursive avec le paquetage `Fib`, car elle appelle `Fib.Eval`. Plus précisément, `DeplieFib` est obtenue à partir de la version récursive naïve de Fibonacci en remplaçant les appels récursifs par des appels à `Fib.Eval`. Plus généralement, on ne fait jamais référence à `DeplieFib` dans le programme en dehors de la création du paquetage `Fib` : on utilise `Fib.Eval` à la place. Par exemple, le programme ci-dessous affiche le graphe des appels pour 6.

```
with Memoize ;

procedure Fibonacci is

   function DeplieFib(N: Natural) return Natural ;

   package Fib is new Memoize(DeplieFib) ;

   function DeplieFib(N: Natural) return Natural is
   begin
      if N <= 1 then
         return 1 ;
      else
         return Fib.Eval(N-2)+Fib.Eval(N-1) ;
      end if ;
   end ;

   R: Natural := Fib.Eval(6);
begin
   Fib.GrapheAppels ;
end ;
```

On attend un affichage de la forme :

```
 0: 1
```

```
1: 1
2: 2
3: 3
4: 5
5: 8
6: 13
```

On demande ici d'utiliser ce paquetage pour écrire un programme qui affiche tous les résultats des fonctions suivantes pour `N` variant de 0 à 10000 :

1. pour la fonction `Fybracuse` définie par

```
function Fybracuse(N: Natural) return Natural is
begin
   if N <= 1 then
      return 1 ;
   elsif N mod 4 = 0 then
      return Fybracuse(N/4)+Fybracuse(N/2) ;
   elsif N mod 2 = 0 then
      return Fybracuse(N/2)+1 ;
   else
      return Fybracuse(3*N+1)/2 ;
   end if ;
end ;
```

2. pour la fonction `Mad` définie par

```
function Mad(N:Natural) return Natural is
begin
   if N=997 then
      return 1004 ;
   elsif N > 997 then
      return Mad(Mad(N-1)-7)+1 ;
   else
      return Mad(Mad(N+1)+7)-1 ;
   end if ;
end ;
```

4 Travail à faire

Chaque équipe doit déposer sur Teide **avant le vendredi 25 mars 2011 à 17h** une archive `tar.gz` contenant :
- Les sources complètes du TP (y compris les sources fournies et les éventuels pilotes de tests développés pour l'occasion).
- Un rapport au format **pdf** de 4 pages maximum présentant :
  1. brièvement, l'état d'avancement du TP et le contenu de l'archive ;
  2. éventuellement, quelques explications sur l'implémentation, à condition qu'elles ne figurent pas déjà dans le sujet, et qu'elles semblent nécessaire pour comprendre les sources commentées ;
  3. une présentation détaillée de la démarche de validation (tests de correction) et d'expérimentation (tests de performances).

Dans la suite de cette section, on vous détaille la liste des fichiers fournis en indiquant ceux que vous devez modifier/compléter. Vous pouvez bien sûr étendre les paquetages demandés en ajoutant de nouvelles procédures ou fonctions.

Essentiellement, vous devez implémenter deux versions du dictionnaire : une version avec des arbres non équilibrés, et une version avec des arbres AVL. On vous demande ensuite de comparer les coûts en temps induits par ces deux versions sur les applications demandées :
- Pour la base de données des codes postaux, on demande de mesurer le coût en temps de l'initialisation de la base de donnée avec l'ensemble des codes postaux.
- Pour la memoïzation, on mesurera les temps de calcul des 10001 premières valeurs de `Fybracuse` et `Mad` (voir section 3.2).

Ce TP étant *très* long, vous n'êtes pas obligés de tout traiter. Il est donc recommandé de traiter les tâches ci-dessous dans l'ordre et ne passant à la tâche suivante que lorsque la validation de la tâche courante est complètement achevée.

4.1 Dictionnaire générique

Pour le dictionnaire générique, on vous fournit les fichiers suivants :
- L'interface `comparaisons.ads` et l'implémentation `comparaisons.adb` du paquetage `Comparaisons` qui définit le type `Comparaison` introduit section 1. Ce paquetage définit aussi des fonctions `Compare` à utiliser dans ce TP.
- L'interface `dictionnaire.ads` du paquetage générique `Dictionnaire` présenté section 2.1.
- Un embryon d'implémentation pour le fichier `dictionnaire.adb`. Vous devez donc modifier/compléter ce fichier à l'aide des indications de la section 1.

On vous conseille de commencer par réaliser et **tester** complètement une version avec des arbres de recherche non équilibrés avant de passer à la version équilibrée.

4.2 Base de données des codes postaux

On vous fournit les fichiers suivants :
- L'interface `parcourircodespostaux.ads` et l'implémentation `parcourircodespostaux.adb` de la procédure générique `ParcourirCodesPostaux` (voir section 2.2). On vous en fournit aussi un exemple d'utilisation `testp.adb` qui permet de vérifier qu'un fichier respecte le format attendu.
- L'interface `ensstring.ads` et l'implémentation `ensstring.adb` du paquetage `EnsString` (voir section 3.1).
- Un embryon d'implémentation pour le fichier `codespostaux.adb` qui est le programme d'interrogation de la base de données des codes postaux. Vous devez donc modifier/compléter ce fichier (voir

section 3.1).

4.3 Mémoïzation

On vous fournit les fichiers suivants :
– L'interface `memoize.ads` du paquetage générique `Memoize` (voir section 3.2).
– Le fichier `fibonacci.adb` qui illustre comment on peut utiliser ce paquetage pour calculer Fibonacci.
– Un embryon d'implémentation pour le fichier `memoize.adb`. Vous devez donc modifier/compléter ce fichier.