Examen du cours IN303

On définit un *pré-tas* comme étant un arbre dont les nœuds sont étiquetés par des entiers et ont deux fils, un fils gauche et un fils droit :

```
Inductive pretas : Set := V : pretas | B : pretas -> nat -> pretas -> pretas.
```

`V` est le pré-tas vide, et `(B g n d)` est le pré-tas de fils gauche `g`, de fils droit `d` et de valeur `n`. Les pré-tas servent de structure de stockage des valeurs entières à chaque nœud.

Un *tas* est un pré-tas vide ou bien un pré-tas `(B g n d)` de valeur `n` inférieure ou égale à toutes les valeurs stockées dans `g` ou `d`, et tel que `g` et `d` sont des tas.

Le but de l'exercice est de formaliser certaines propriétés des tas; une des applications en est le *tri par tas* (heapsort).

1. Le type `pretas` est inductif, ce qui signifie que l'ensemble des pré-tas est le plus petit tel que `V` soit un pré-tas et `(B g n d)` est un pré-tas pour tout entier `n` dès que `g` et `d` sont des pré-tas. Ceci est représenté en Coq par un principe de récurrence. Dites lequel :

   (A)
   ```
   (P:liste->Prop) (P nil) -> ((l:liste) (n:nat) (P l) -> (P (cons n l)))
                             -> (l:liste) (P l)
   ```
   (B)
   ```
   (P:pretas->Prop) (P V) ->
                           ((g:pretas)  (P g) -> (n:nat) (d:pretas) (P d)
                                 -> (P (B g n d)))
                             -> (t:pretas) (P t)
   ```
   (C)
   ```
   (P:pretas->Prop) (P V) ->
                           ((d:pretas) (n:nat) (P d) -> (g:pretas) (P (B g n d)))
                             -> (t:pretas) (P t)
   ```

2. On veut écrire un prédicat `minore` tel que `(minore n t)` soit vrai si et seulement si l'entier `n` est inférieur ou égal à tous les entiers stockés dans le pré-tas `t`. Parmi les possibilités suivantes, quelle est la seule bonne ? (On suppose comme dans l'exemple du tri par insertion qu'on dispose d'une fonction `au_plus : nat -> nat -> bool` telle que `(au_plus n m)` est `true` ssi $n \leq m$.)

   (A)
   ```
   Fixpoint minore [n:nat; t:pretas] : Prop :=
       Cases t of
           V => True
         | (B g m d) => (au_plus n m)=true /\ (minore n g) /\ (minore n d)
       end.
   ```
   (B)
   ```
   Fixpoint minore [n:nat; t:pretas] : Prop :=
       Cases t of
           V => False
   ```

```
      | (B g m d) => (au_plus n m)=true /\ (minore n g) /\ (minore n d)
    end.
```

(C) `Fixpoint minore [n:nat; t:pretas] : Prop :=`

```
    Cases t of
        V => False
      | (B g m d) => (au_plus n m)=true /\ (minore n t)
    end.
```

3. On veut maintenant définir un prédicat `tas` tel que `(tas t)` soit vrai ssi le pré-tas `t` est un tas. Lequel proposez-vous ?

   (A) `Fixpoint tas [t:pretas] : Prop :=`

   ```
       Cases t of
           V => True
         | (B g n d) => (minore n t) /\ (tas g d)
   ```

   (B) `Fixpoint tas [t:pretas] : Prop :=`

   ```
       Cases t of
           V => True
         | (B g n d) => (minore n g) /\ (tas g) /\ (minore n d) /\ (tas d)
       end.
   ```

   (C) `Fixpoint tas [t:pretas] : bool :=`

   ```
       Cases t of
           V => True
         | (B g n d) => (minore n t) /\ (tas g) /\ (tas t)
       end.
   ```

4. Quelle est la bonne lecture de la ligne suivante ?

   ```
   Lemma choix_bool : (b:bool) b=true \/ b=false.
   ```

   (A) `b` est un booléen qui est soit vrai soit faux.

   (B) Tout booléen est vrai ou alors `b` est faux.

   (C) Tout booléen est soit vrai soit faux.

5. Montrer informellement le lemme :

   ```
   Lemma ajoute_minore : (t:pretas) (m,n:nat) (au_plus m n)=true -> (minore m t)
                               -> (minore m (ajoute n t)).
   ```

   Vous pourrez vous aider des lemmes auxiliaires suivants, sans avoir à les démontrer :

   ```
   Lemma choix_bool : (b:bool) b=true \/ b=false.
   Lemma au_plus_trans : (m,n,k:nat) (au_plus m n)=true -> (au_plus n k)=true ->
                                 (au_plus m k)=true.
   ```

   Vous préciserez soigneusement quels dispositifs logiques vous utilisez : récurrences [sur quoi], analyses de cas, lemmes utilisés, etc.

6. Montrer, de la même façon, le lemme :

```
Lemma au_plus_minore : (t:pretas) (m,n:nat) (au_plus m n)=true -> (minore n t)
                                -> (minore m t).
```

7. On définit une fonction `ajoute` telle que `(ajoute n t)` ajoute un entier `n` au tas `t` comme suit :

```
Fixpoint ajoute [n:nat; t:pretas] : pretas :=
  Cases t of
      V => (B V n V)
    | (B g m d) => if (au_plus n m)
                   then (B (ajoute m d) n g)
                   else (B (ajoute n d) m g)
  end.
```

   Montrer informellement que :

```
Lemma ajoute_preserve_tas : (t:pretas) (tas t) -> (n:nat) (tas (ajoute n t)).
```

   Vous pourrez vous aider des lemmes démontrés précédemment, ainsi que du lemme auxiliaire suivant que vous n'aurez pas besoin de démontrer :

```
Lemma au_plus_total : (m,n:nat) (au_plus m n)=false -> (au_plus n m)=true.
```

8. (Question bonus.) Spécifier un prédicat `memes_elements   : liste -> pretas -> Prop` disant qu'une liste d'entier et un pré-tas contiennent les mêmes éléments, où :

```
Inductive liste : Set := nil : liste | cons : nat -> liste -> liste.
```

   et montrer que la fonction de conversion d'une liste en tas :

```
Fixpoint conversion [l:liste] : pretas :=
  Cases l of
      nil => V
    | (cons n l') => (ajoute n (conversion l'))
  end.
```

   produit d'une part un tas :

```
Lemma conversion_produit_tas : (l:liste) (tas (conversion l)).
```

   et d'autre part produit un tas ayant mêmes éléments que la liste de départ :

```
Lemma conversion_memes_elements : (l:liste) (memes_elements l (conversion l)).
```

L'idée du tri par tas est de procéder comme suit : soit `l` une liste d'entiers à trier. Le tas résultant `(conversion l)`, étant un tas, possède à sa racine l'élément minimum de la liste `l` (à moins que `l` ne soit vide). Il ne reste plus qu'à éliminer la racine de ce tas pour obtenir le tas des éléments restants de `l`; en itérant ce processus, on obtient tous les éléments de `l` en ordre croissant.