FEUILLE D'EXERCICES N°3 DE L'OPTION D'INFORMATIQUE.

Le but est d'étudier diverses organisations pour un «dictionnaire»; un dictionnaire étant un ensemble de mots ou de clefs sur lequel on peut effectuer deux opérations :

i rechercher l'existence d'une clé dans l'ensemble déjà défini

ii insérer (ou ajouter) une nouvelle clé dans cet ensemble

En notre sens un dictionnaire est donc un ensemble de clés qui évolue dans le temps au gré des insertions.

1 Représentation sous forme de liste

Les clés sont rangées dans l'ordre d'insertion dans une liste référencée `dico` dont on précisera le type. L'insertion d'une nouvelle clé dans un dictionnaire de longueur **p** se fait en fin de liste.

1. Écrire une fonction **vérif** qui vérifie qu'un dictionnaire n'a pas 2 clés identiques.

2. Écrire une fonction **nettoie** qui supprime les doublons dans un dictionnaire et ne garde qu'une seule clé parmi les clés identiques.

3. Écrire une fonction booléenne **contient** qui effectue la recherche d'une clé dans un dictionnaire et dont le résultat est `true` si la clé est dans le dictionnaire et `false` sinon.

4. Écrire une fonction **insère** qui ajoute une clef à un dictionnaire si la clef n'est pas déjà dans le dictionnaire.

2 Représentation triée

Dans cette partie on suppose que la liste des clés dans le dictionnaire est rangée en ordre croissant (selon la fonction $<$ polymorphe de **caml** dans un tableau de longueur fixe. On supposera que le dictionnaire est de taille fixe **tmax**).

5. Écrire une fonction **vérif2** qui vérifie qu'un dictionnaire n'a pas 2 clés identiques et que les clés sont bien rangées en ordre croissant.

6. La recherche d'une clé **c** peut se faire de manière plus efficace qu'en **1** en procédant par dichotomie, comme suit (p est la taille du dictionnaire) :

- comparer la clé c à la clé de rang $E\left(\frac{p}{2}\right)$ soit d;
- Si $c = d$ la recherche est terminée
  Si $c < d$ recommencer entre 1 et $E\left(\frac{p}{2}\right) - 1$
  Si $c > d$ recommencer entre $E\left(\frac{p}{2}\right) + 1$ et p
- Il y a échec (c'est-à-dire recherche infructueuse) si la méthode conduit à recommencer sur un intervalle vide. On remarquera que de façon générale la recherche sur l'intervalle $[q, r]$ conduit à comparer la clé c à la clé de rang $E\left(\frac{q+r}{2}\right)$

Écrire une fonction ayant deux paramètres résultats `succès` et `rang` et type booléen et entier, telle que :

- si la clé c existe dans le dictionnaire, `succès = vrai` et `rang =` le rang de c
- sinon `succès = faux` et `rang =` le rang de la clé du dictionnaire immédiatement inférieure à c

7. Écrire une fonction **insère2** qui ajoute une clef à un dictionnaire si la clef n'est pas déjà dans le dictionnaire tout en conservant l'ordre (il faudra amener cette clé à la bonne place dans le tableau trié).

3 Représentation arborescente

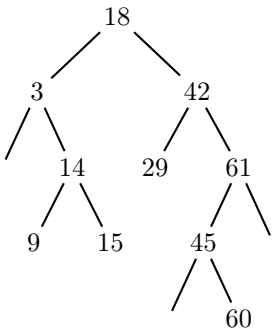
La méthode précédente a l'inconvénient d'obliger à décaler toutes les clés placées à droite de celle que l'on veut insérer; nous décrivons maintenant une méthode qui évite ce décalage et de plus élimine la contrainte de taille.

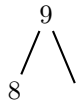
Le dictionnaire est représenté par un arbre binaire dont les sommets contiennent comme information une clé et ont deux successeurs : fils gauche et fils droit. Un sommet peut éventuellement être vide : il n'a alors aucun successeurs.

L'arbre vérifie de plus la propriété suivante fondamentale : appelons descendants d'un sommet **s**, soit ses successeurs soit les descendants de ses successeurs; les descendants gauches (droits) de **s** sont soit son successeur gauche (droit) soit les descendants de ce successeur.

L'arbre est tel que pour tout sommet contenant une clé **c**, les clés des descendants gauches sont inférieurs à **c** et les clés des descendants droits sont supérieurs à **c**.

exemple : l'ensemble des clés est ici $\mathbb{N}$; les sommets vides ne sont pas représenté; la suites des clés insérées est la suivante : 18, 42, 3, 14, 29, 61, 15, 45, 60, 9. L'arbre obtenu est le suivant :



L'adjonction de la clé 8 conduit à modifier la feuille 9 en 

8. Écrire une fonction booléenne de recherche d'une clé dans un tel dictionnaire.

9. Écrire une procédure `insère3` d'insertion d'une nouvelle clé.

10. Écrire une procédure `ajoute3` de construction d'un tel arbre à partir d'une liste de clés.

11. Écrire une fonction `parcours` qui parcourt l'arbre de façon à obtenir une liste triée de ses clés.

12. Écrire une fonction qui supprime une clé dans un arbre tout en maintenant la structure de dictionnaire.

13. Écrire une fonction `profondeur` qui, étant donné un arbre-dictionnaire et un mot, retourne la profondeur à laquelle se trouve le mot (on retournera -1 si le mot ne se trouve pas dans l'arbre et 0 si le mot est à la racine de l'arbre).

14. Écrire une fonction `est_descendant_de` qui, étant donné un arbre-dictionnaire *tree* et deux mots m_1 et m_2, retourne `true` si m_2 est dans l'un des sous-arbres issu du nœud contenant le mot m_1 dans l'arbre *tree* et `false` sinon.

15. Écrire une fonction `ancetre` qui, étant donné un arbre *tree*, et deux mots $m1$ et $m2$ retourne l'ancêtre commun à $m1$ et $m2$ le plus profond dans l'arbre *tree* (on affichera un message d'erreur si l'un des deux mots n'est pas dans l'arbre dictionnaire *tree*).

16. Écrire une fonction `verif_dico` qui, étant donné un arbre *tree*, retourne `true` si *tree* est un arbre-dictionnaire et `false` sinon.