Algorithmique
Contrôle n° 1

INFO-SPÉ – EPITA

D.S. 64319.5 BW (8 nov 2010 - 10 :00)

Consignes (à lire) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- ☐ **Les algorithmes :**
  - Tout algorithme doit être écrit dans le langage ALGO (pas de C, CAML ou autre).
  - Tout code ALGO non indenté ne sera pas corrigé.
  - Tout ce dont vous avez besoin (types, routines) est indiqué en **annexe** (dernière page) !
- ☐ Durée : 2h00

Exercice 1 (Hachages – *7 points*)

Nous avons déjà constaté que la suppression générait un problème de réorganisation des données. Le moyen de ne pas avoir à le faire est d'introduire un état supplémentaire aux cases du tableau de hachage. Jusqu'à présent, les cases étaient *"vide"* ou *"occupée"*. Nous allons gérer un troisième état qui est *"libre"*.

A l'initialisation du tableau les cases sont toutes *"vide"*. Lorsque l'on ajoute un élément dans une case, cette dernière devient *"occupée"*. Enfin lorsque l'on supprime un élément dans une case, celle-ci devient *"libre"*. *Une fois "occupée", une case ne pourra jamais redevenir "vide".*

Nous pourrions alors utiliser trois fonctions qui seraient :

* ***estVide*** (fonction booléenne retournant vrai si une case est vide),
* ***estOccupee*** (fonction booléenne retournant vrai si une case est occupée),
* ***estLibre*** (fonction booléenne retournant vrai si une case est libre).

Mais nous allons leur préférer une fonction ***etatEst*** qui teste les trois états.

```
Algorithme Fonction etatEst : Booléen
Paramètres locaux
    t_hachage  th
    entier     i
    typeEtat   etat
Debut
    Retourne(th[i].etat=etat)
Fin Algorithme Fonction etatEst
```

Cette fonction utilise les déclarations suivantes :

```
Constantes
    m = ...                              /* taille du tableau de hachage */
Types
    typeEtat  = (vide, libre, occupee)   /* Les différents états sont définis là */
    t_element = ...                      /* Le type des éléments est défini là */
    t_elt     = Enregistrement
        typeEtat  etat
        t_element elt
        entier    lien
    Fin Enregistrement t_elt
    t_hachage = m t_elt
Variables
    t_hachage   th
```

Nous supposerons la fonction de hachage h, dont le prototype est le suivant, définie.

```
Algorithme fonction h : entier
Paramètres locaux
    t_element  x
Début
    /* Retourne la valeur de hachage primaire de l'élément x */
Fin algorithme fonction h
```

1. Dans le cas du hachage coalescent et en tenant compte de la possibilité de supprimer un élément, donner le principe de l'algorithme d'une fonction booléenne d'ajout d'un élément. Cette fonction doit renvoyer $Faux$ si le tableau est plein et que l'ajout n'a pas pu être effectué et $Vrai$ dans tous les autres cas.

2. En utilisant les déclarations précédentes écrire l'algorithme de la fonction booléenne *ajouter_HCO* correspondante au principe précédent. Pour les besoins de cet algorithme, le tableau de hachage sera indicé de 0 à $m-1$.

Exercice 2 (Arbre 2-3-4 : insertions – *4 points*)

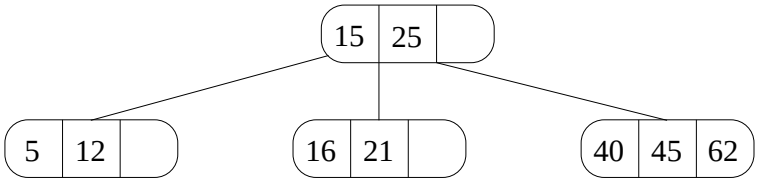

FIGURE 1 – Arbre 2.3.4 pour ajouts

Insérer **successivement** les clés 18, 42 puis 23 dans l'arbre 2.3.4. de la figure 1 en appliquant le principe de l'insertion avec éclatement à la descente.

Dessiner **uniquement** l'arbre obtenu après chaque insertion en indiquant clairement, s'il y a lieu, le ou les nœud(s) ayant subi un éclatement.

Exercice 3 (Arbre 2-3-4 : Des croissants – *4 points*)

Écrire un algorithme qui construit une chaîne de caractères représentant la liste des clés (de type `chaine`) contenues dans un arbre 2.3.4 en ordre décroissant.

Remarque : La chaîne construite présentera les clés séparées par des virgules.

Exercice 4 (Arbres généraux : Préfixe - Suffixe – *7 points*)

Dans cet exercice, on se propose de remplir un vecteur avec les clefs des nœuds d'un arbre général. On place chaque nœud **deux** fois dans le vecteur : lors de la première rencontre (ordre préfixe) et lors de la *dernière* rencontre (ordre suffixe) du parcours profondeur.

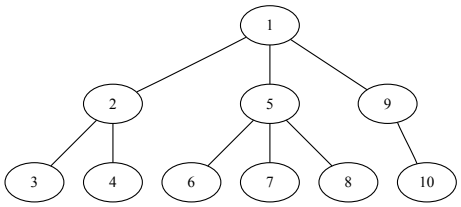

FIGURE 2 – Un arbre général

Le tableau suivant représente le remplissage du tableau pour l'arbre de la figure 2 :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	2	3	3	4	4	2	5	6	6	7	7	8	8	5	9	10	10	9	1

1. Écrire la procédure `ps_nuplet(A,c,v)` qui parcourt l'arbre général `A` (en représentation nuplets de pointeurs) en profondeur et remplit le vecteur de clefs en respectant l'ordre décrit (chaque clef est placée dans le tableau en préfixe, puis en suffixe). L'entier `c` indique la position courante dans le vecteur.

2. Écrire la fonction `remplissage_nuplet(A,v)` qui remplit le vecteur `v` avec les clefs de l'arbre `A` en utilisant la procédure `ps_nuplet(A,c,v)`. Cette fonction devra renvoyer **la taille** de l'arbre.

3. Écrire à nouveau la procédure de la question 1) mais en représentation premier-fils/frère-droit. Nous appellerons cette procédure `ps_dyn(A,c,v)`.

Annexes

Type de données représentant les arbres 2-3-4 :

```
constantes
    degre = 2
types
    /* déclaration du type t_ element */
    t_a234    = ↑ t_noeud_234
    tab3cles  = (2*degre-1) chaine
    tab4fils  = (2*degre) t_a234
    t_noeud_234 = enregistrement
        entier    nbcles
        tab3cles  cle
        tab4fils  fils
    fin enregistrement t_noeud_234
```

Arbres Généraux en représentation nuplets

```
constantes
  Max =  /*une valeur suffisante !*/
types
    t_element = ...
    t_arbre_nuplets = ↑t_noeud_nuplets

    t_tab_fils = Max t_arbre_nuplets

    t_noeud_nuplets = enregistrement
      t_element         cle
      entier            nbFils
      t_tab_fils        fils
    fin enregistrement t_noeud_nuplets
```

Arbres Généraux en représentation premier-fils/frère-droit

```
types
    t_element = ...
    t_arbre_dyn = ↑t_noeud_dyn

    t_noeud_dyn = enregistrement
      t_element         cle
      t_arbre_dyn       fils, frere
    fin enregistrement t_noeud_dyn
```

Vecteurs de clefs

```
constantes
    MaxVect =  /*une valeur suffisante !*/
types
    t_element = ...
    t_vect_cles = MaxVect t_element
```