Algorithmique
Correction Contrôle nº 3 (C3)

Info-spé - S3# – Epita

5 mars 2019 - 14 : 45

Solution 1 **(Hachage linéaire –** ***2 points*****)**

Représentation des structures de données dans le cas du hachage linéaire *(Hachage linéaire avec un coefficient de décalage* $d = 5$*) voir tableau 1 :*

Table 1 – Hachage linéaire

0	aragog
1	buck
2	crockdur
3	croutard
4	dobby
5	fumseck
6	
7	hedwige
8	kreattur
9	nagini
10	missteigne

Solution 2 **(Quelques questions –** ***5 points*****)**

1. Les trois propriétés essentielles que doit posséder une fonction de hachage sont :
   (a) Uniforme
   (b) Déterministe
   (c) Facile et rapide à calculer
2. Le hachage coalescent).
3. Le double hachage permet de résoudre le phénomène de regroupement ou accumulation d'éléments (clustering) provoqué par le hachage linéaire.
4. Les méthodes de hachage de base sont : extraction, complétion, division, multiplication.
5. Le hachage linéaire ou le double hachage.
6. Le hachage avec chaînage séparé. En effet les éléments sont chaînés entre eux à l'extérieur du tableau.

Solution 3 **(Sérialisation – *5 points*)**

1. *Le vecteur de pères :*

0	1	2	3	4	5	6	7	8	9	10	11
2	2	10	6	2	10	7	8	10	2	−1	6

2. **Spécifications :**
   La fonction `buildParentVect(`T`, `n`)` retourne le vecteur (représenté par une liste en Python) de pères correspondant à l'arbre T en **implémentation "premier fils - frère droit"** de taille n. Les clés de l'arbre T sont les entiers dans $[0, n[$ (sans redondance).

```
            # Recursively fills P with "parent" relations from T
            def __buildParentVect(T, P):
                C = T.child
                while C != None:
                    P[C.key] = T.key
                    __buildParentVect(C, P)
                    C = C.sibling

            def buildParentVect(T, n):
                P = [-1] * n
                __buildParentVect(T, P)
                return P
```

Solution 4 **(Croissants – *4 points*)**

Spécifications :
`BtreeToList(`B`)` retourne la liste des clés du B-arbre B en ordre croissant.

```
                def __BtreeToList(B, L):
                    if B.children == []:
                        for i in range(B.nbKeys):    # L += B.keys
                            L.append(B.keys[i])
                    else:
                        for i in range(B.nbKeys):
                            __BtreeToList(B.children[i], L)
                            L.append(B.keys[i])
                        __BtreeToList(B.children[B.nbKeys], L)

                def BtreeToList(B):
                    L = []
                    if B:
                        __BtreeToList(B, L)
                    return L
```

Solution 5 **(Mesure sur les B-arbres – *4 points*)**

Spécifications :
`occupation(`B`)` retourne le nombre moyen de clés par nœud du B-arbre B.

```
        def __occupation(B):        # returns the pair (nb nodes, nb keys)
            (k, n) = (B.nbkeys, 1)
            for C in B.children:
                (kc, nc) = __occupation(C)
                k += kc
                n += nc
            return (k, n)

        def occupation(B):
            if not B:
                return 0
            else:
                (k, n) = __occupation(B)
                return (k/n)
```