Algorithmics
Correction Midterm #3 (C3)

Undergraduate 2^{nd} year - S3# – Epita

5 mars 2019 - 14 : 45

Solution 1 **(Linear probing – *2 points*)**

Collision resolution using the linear probing principle with an offset coefficient $d = 5$:

Table 1: Linear probing

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

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

1. The three essential properties required of a hash function are:
   (a) Uniform
   (b) Consistent
   (c) Easy and fast to compute
2. The secondary collisions appear with the coalesced hashing.
3. The double hashing allows to solve the phenomenon of clustering generated by the linear probing.
4. The basic hashing methods are: extraction, compression, division, multiplication.
5. The linear probing or the double hashing.
6. The hashing with separate chaining. The elements are chained together outside the hash table.

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

1. *The parent vector:*

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. **Specifications:**
   The function `buildParentVect(`T`, `n`)`, from the tree T of size n, fills and returns the corresponding parent vector (represented as a list in Python), using **left child - right sibling implementation**. Keys in the tree T are integers in $[0, n[$ (without repetition).

```
            # 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* (Ascending – *5 points*)**

Specifications:
`BtreeToList(`B`)` returns the list of the keys of the B-tree B in increasing order.

```
def __BtreeToList(B, L):
    if B.children == []:
    # L += B.keys or
        for i in range(B.nbkeys):
            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* (B-tree measures – *4 points*)**

Specifications:
`occupation(`B`)` returns the list of the keys of the B-tree B in increasing order.

```
        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)
```