Algorithmics
Correction Midterm #3 (C3)

Undergraduate 2^{nd} year - S3 – Epita

29 October 2018 - 13 : 30

Solution 1 **(Hashing Strongly Connected –** ***4 points*****)**

1. The linear probing or the double hashing.

2. The hashing with separate chaining. Le hachage avec chainage séparé. the elements are chained together outside the hash table.

3. The search by interval is incompatible with the hashing due to the dispersion of thes elements.

4. The secondary collisions appear with the coalesced hashing.

5. The directed graph `G=<S,A>` defined by:

```
   S={1,2,3,4,5,6,7,8,9,10}
et A={(1,2),(1,6),(1,7),(2,3),(2,6),(3,1),(3,5),(4,3),(4,8),(4,9),(4,10),
      (5,1),(7,6),(8,5),(8,10),(10,9)}
```

   is that of figure 1

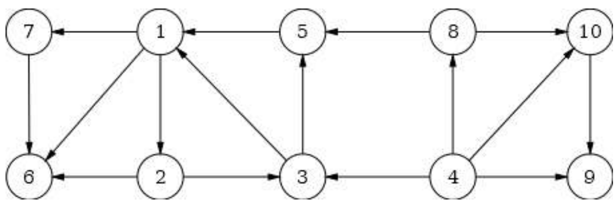

Figure 1: Directed graph.

6. The Indegree table is as follows:

	1	2	3	4	5	6	7	8	9	10
InDegree	2	1	2	0	2	3	1	1	2	2

Solution 2 **(Equality –** ***5 points*****)**

Specifications:

The function `same(T, B)` tests whether T, a general tree in "classical" representation, and B, a general tree in *first child - right sibling* representation, are identical.

```
# with return statement in loop
    def equal(T, B):
        if T.key != B.key:
            return False
        else:
            Bchild = B.child
            for Tchild in T.children:
                if Bchild == None or not(equal(Tchild, Bchild)):
                    return False
                Bchild = Bchild.sibling
            return Bchild == None

# without return in the loop
def equal2(T, B):
    if T.key != B.key:
        return False
    else:
        Bchild = B.child
        i = 0
        while i < T.nbChildren and (Bchild and equal2(T.children[i], Bchild)):
            i += 1
            Bchild = Bchild.sibling
        return i == T.nbChildren and Bchild == None
```

Solution 3 **(Levels –** ***4 points*****)**

Specifications:

The function `levels(T)` builds a list of the keys of T level by level.

```
    def levels(T):
        q = queue.Queue()
        q.enqueue(T)
        q2 = queue.Queue()
        Levels = []
        L = []
        while not q.isempty():
            T = q.dequeue()
            L.append(T.key)
            C = T.child
            while C:
                q2.enqueue(C)
                C = C.sibling
            if q.isempty():
                (q, q2) = (q2, q)
                Levels.append(L)
                L = []

        return Levels
```

Solution 4 **(Maximum Gap –** ***4 points*****)**

Specifications:

The function `maxgap(`B`)` computes the maximum gap of the B-tree B.

```
# optimised version: searching in all children is useless,
# first and last child are sufficient!

    def __maxgap(B):
        gap = 0
        for i in range(B.nbkeys-1):
            gap = max(gap, B.keys[i+1] - B.keys[i])
        if B.children:
            gap = max(gap, __maxgap(B.children[0]))
            gap = max(gap, __maxgap(B.children[-1]))
        return gap

# less optimized...

    def __maxgap2(B):
        gap = 0
        for i in range(B.nbkeys-1):
            gap = max(gap, B.keys[i+1] - B.keys[i])

        for child in B.children:
            gap = max(gap, __maxgap2(child))
        return gap

    def maxgap(B):
        return 0 if B is None else __maxgap(B)
```

Solution 5 **(B-Trees and Mystery –** ***3 points*****)**

1. Application results:

	Returned result	*Call number*
(a) `mystery(`B_1`, 1, 77)`	29	10
(b) `mystery(`B_1`, 10, 30)`	11	7

2. `mystery(B, a, b)` (`a` $<$ `b`) computes the number of values of B in $[a, b[$.