Algorithmics
Midterm #3 (C3)

Undergraduate 2nd year (S3)
Epita

24 October 2016 - 14 : 45

Instructions (read it) :

□ You must answer on **the answer sheets provided.**

- No other sheet will be picked up. Keep your rough drafts.
- Answer within the provided space. **Answers outside will not be marked**: Use your drafts!
- Do not separate the sheets unless they can be re-stapled before handing in.
- Penciled answers will not be marked.

□ The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.

□ **Code:**

- All code must be written in the language `Python` (no C, Caml, Algo or anything else).
- **Any `Python` code not indented will not be marked.**
- All that you need (classes, types, routines) is indicated where needed!
- You can write your own functions as long as they are documented (we have to known what they do).
  In any case, the last written function should be the one which answers the question.

□ Duration : 2h

LAO-TSEU S'EST ÉPUISÉ À LE DIRE: CELUI QUI MÉPRISE L'IMPROBABLE EST COMME LA CHENILLE QUI, S'ÉLANÇANT À TRAVERS LE SENTIER, REFUSE DE CROIRE QU'ON EN AURA FAIT UNE AUTOROUTE AVANT QU'ELLE N'ATTEIGNE L'AUTRE CÔTÉ!
ET SPROUTCH DONC.

Some Hashing

Exercise 1 (Linear probing – *2 points*)

Assume the following set of key E={data, kirk, neelix, odo, picard, q, quark, sisko, tuvok, worf} and the table 1 of hash values associated with each key of this set E. These values are lying between 0 and 10 ($m = 11$).

Table 1: Hash values

data	4
kirk	5
neelix	3
odo	1
picard	7
q	6
quark	2
sisko	7
tuvok	1
worf	7

Present the collision resolution for adding all the keys of the set E in the order of the table 1 (from `data` to `worf`) using the linear probing principle with an offset coefficient $d = 4$

Exercise 2 (Hashing: Valid tables – *3 points*)

Suppose the keys from A to G with the hash values given in the table 2.

Table 2: Hash values

keys	A	B	C	D	E	F	G
h(keys) m=7	2	0	0	4	4	4	2

If they are inserted in any order, on the principle of linear probing (with $d = 1$), in an initially empty table of size 7, which of the following tables may not result in the insertion of these keys?

Table 3: Possible tables ?

	0	1	2	3	4	5	6
Table (A)	C	G	B	A	D	E	F
Table (B)	F	G	B	D	A	C	E
Table (C)	B	C	A	G	E	D	F
Table (D)	G	E	C	A	D	B	F

Exercise 3 (Hashing: Questions. . . – *3 points*)

1. Name three properties required of a hash function.
2. What is the cause of a secondary collision ?
3. Collisions set apart, what phenomenon is caused by the linear probing and what do we envisage to solve it ?

Some Trees

The (general) trees we work on are the same as the ones in tutorials.

Classical implementation

```
class Tree:
    def __init__(self, key=None, children=None):
        self.key = key
        if children is not None:
            self.children = children
        else:
            self.children = []

    @property
    def nbChildren(self):
        return len(self.children)
```

First child - right sibling implementation

```
class TreeAsBin:
    def __init__(self, key, child=None, sibling=None):
        self.key = key
        self.child = child
        self.sibling = sibling
```

Exercise 4 (Average Arity of a General Tree – *4 points*)

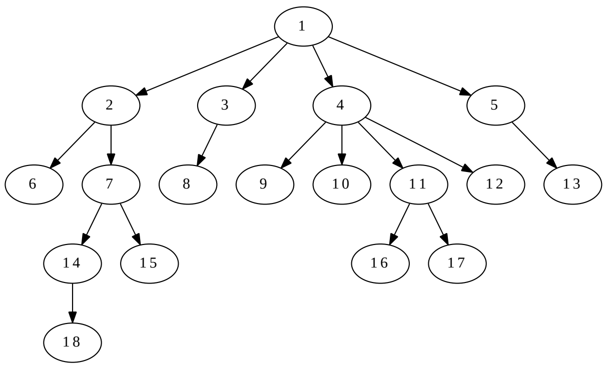

Figure 1: General tree

We now study the average arity (number of children for a node) in a general tree. We define the average arity as the sum of the number of children per node divided by the number of internal node (without leaves).

For example, in the tree from figure 1, there's 8 internal nodes and the sum of the number of children per node is 17 (check the arrows), thus the average arity is: $17/8 = 2.125$.

Write the function `averageArity(`B`)` that returns the average arity of the a general tree T (only one traversal has to be done), for *first child - right sibling* implementation.

Exercise 5 (Equality – *5 points*)

Write the function `same(`T`, `B`)` that tests whether T, a general tree in "classical" representation, and B, a general tree in *first child - right sibling* representation, are identical. That is, they contain same values in same nodes.

A B-Tree

The B-trees we work on are the same as the ones in tutorials.

```python
class BTree:
    degree = None

    def __init__(self, keys=None, children=None):
        self.keys = keys if keys else []
        self.children = children if children else []

    @property
    def nbKeys(self):
        return len(self.keys)
```

Exercise 6 (B-Trees and Mystery – *3 points*)

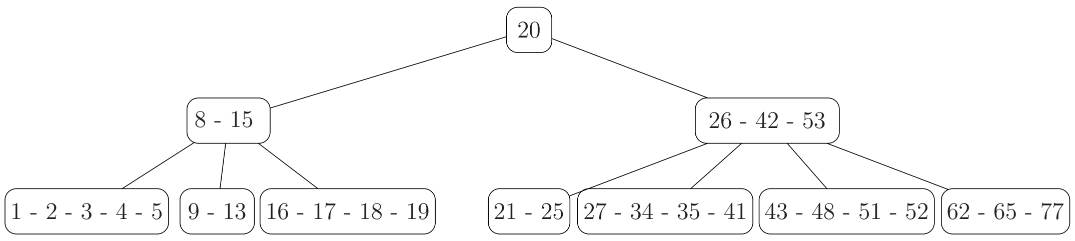

Figure 2: B-tree B_1

```python
def mystery(B, a, b):
    i = 0
    while i < B.nbKeys and B.keys[i] < a:
        i += 1
    c = 0
    if B.children == []:
        while i < B.nbKeys and b > B.keys[i]:
            i += 1
            c += 1
    else:
        c += mystery(B.children[i], a, b)
        while i < B.nbKeys and b > B.keys[i]:
            c += mystery(B.children[i+1], a, b) + 1
            i += 1
    return c
```

1. Let B_1 be the tree in figure 2. For each of the following calls:
   - what is the result?
   - how many calls of `mystery` have been done?

   (a) `mystery(`B_1`, 1, 77)`
   (b) `mystery(`B_1`, 10, 30)`

2. Let `B` be any non-empty B-tree filled with integers, and `a` and `b` two integer values such that `a` $<$ `b`. What does the function `mystery(B, a, b)` calculate?