Algorithmics
Midterm #3 (C3)

Undergraduate 2nd year - S3#
Epita

5 mars 2019 - 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 (class, types, routines) is indicated in the appendix (last page).
  - You can write your own functions as long as they are documented (we have to know what they do).
    In any case, the last written function should be the one which answers the question.
- ☐ Duration : 2h

Exercise 1 (Linear probing – *2 points*)

Assume the following set of key E={aragog, buck, crockdur, croutard, dobby, fumseck, hedwige, kreattur, nagini, missteigne} 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

aragog	0
buck	1
dobby	4
fumseck	0
missteigne	0
nagini	5
crockdur	2
croutard	4
kreattur	8
hedwige	8

Present the collision resolution for adding all the keys of the set E in the order of the table 1 (from `aragog` to `hedwige`) using the linear probing principle with an offset coefficient $d = 5$

Exercise 2 (Some questions – *5 points*)

1. Name three properties required of a hash function.
2. With which collision resolution method do secondary collisions appear?
3. Which hashing method allows to solve the phenomenon of clustering generated by the linear probing?
4. Name two basic hashing methods.
5. Name a hashing method that uses a probe sequence function.
6. Which collision resolution method does not need a hash table whose size is greater or equal than the number of keys to be hashed?

Exercise 3 (Serialization – *C3# - 2019-03*)

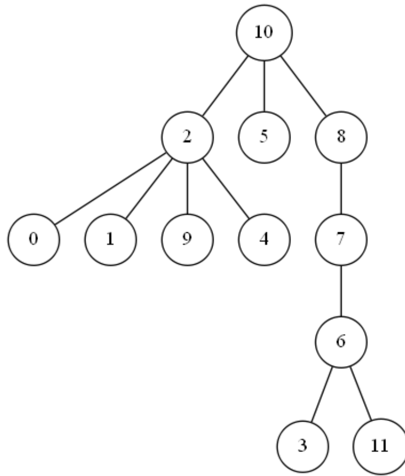

Figure 1: General Tree T

We shall now study an alternative representation for general trees: parent vectors. This implementation is linear and thus can be used to store trees in files (serialization.)

This representation is pretty simple. For each node of the tree, we associate a unique identifier: an integer in the range from 0 to the size of the tree - 1. We then build a vector where the cell i contains the identifier of the immediate parent for the node of identifier i. The parent of the root is -1.

1. Give the parent vector for the tree in figure 1.
2. Write the function `buildParentVect(`T`, `n`)` which, 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).

Exercise 4 (Ascending – *5 points*)

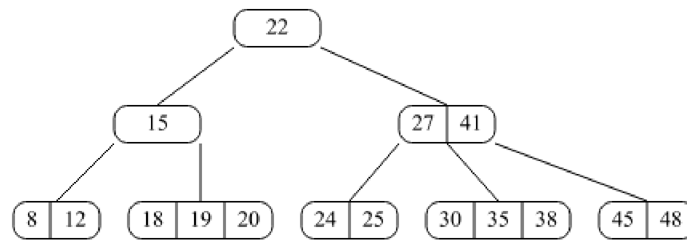

Figure 2: B

Write a function that builds a list of the keys of a B-tree in ascending order.
Application example, with `B` the tree in figure 3:

```
>>> btree2list(B)
[8, 12, 15, 18, 19, 20, 22, 24, 25, 27, 30, 35, 38, 41, 45, 48]
```

Exercise 5 (B-tree measures – *4 points*)

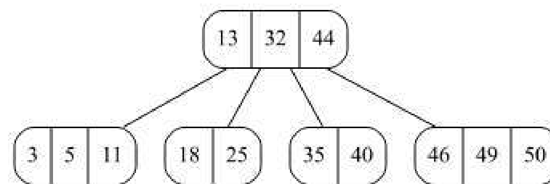

Figure 3: B-Arbre B

In this exercise we intend to measure the *quality* of a B-tree.
Write the function `occupation(B)` that returns the **average number of keys per node** (key number / node number) in the B-tree `B`. The function returns 0 if the tree is empty.
Application example with B the tree in figure 3:

```
>>> occupation(B)
2.6
```

Appendix

Trees

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

Classical implementation

```
class Tree:
    def __init__(self, key, 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
```

B-Trees

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

```
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)
```

Authorised functions and methods

- `len` on lists.
- `range`
- `append` on lists

`Queue`:

- `Queue()` returns a new queue ;
- *q*`.enqueue(`*e*`)` enqueues e in q ;
- *q*`.dequeue()` deletes and returns the first element of q ;
- *q*`.isempty()` tests whether q is empty.

Your functions

You can write your own functions as long as they are documented (we have to know what they do).

In any case, the last written function should be the one which answers the question.