Algorithmics
Midterm #3 (C3)

Undergraduate 2^{nd} year - S3
EPITA

9 November 2021 - 9 : 30

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 (Graphs and components... – *5 points*)

Let `G=<S,A>` be a directed graph defined by:

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

1. Fill-in the array of the indegrees of $G's$ vertices.
2. Give the *preorder* traversal vertices of the graph G starting from the vertex 3 (choose vertices in increasing order).
3. Is the graph G strongly connected ?
4. If NO, how many strongly connected components does it have ?
5. If they exist, which vertices of `G` have a degree equal to 0? If there is none, answer 0.

Exercise 2 (Large Family – *4 points*)

Write the function `morechildren`(T) that checks if each internal node of the tree T has strictly more children than its parent, for *first child - right sibling* implementation.

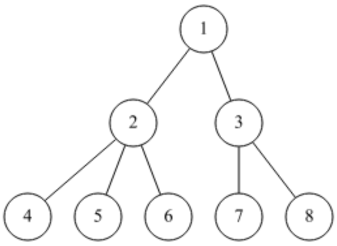

Figure 1: Tree `T1`

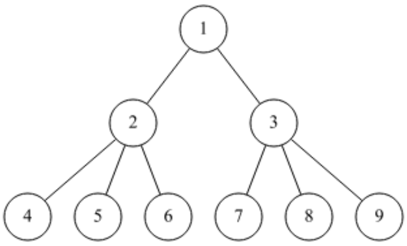

Figure 2: Tree `T2`

Application examples with the trees in figures 1 (`T1`) and 2 (`T2`):

```
    >>> morechildren(T1)
    False
    >>> morechildren(T2)
    True
```

Exercise 3 (Decreasing – *4 points*)

Write the function `decrease`(B) that builds the list of the keys of the B-tree B in decreasing order.
Application example with `B1` the B-tree in figure 3:

```
  >>> decrease(B1)
  [51, 40, 38, 29, 25, 23, 21, 17, 14, 10, 7, 3]
```

Exercise 4 (B-tree: insertions and deletion – *3 points*)

For each question, use the "in going down" principle seen in tutorial (except bonus). Only draw the final tree.

1. Draw the tree resulting from the successive insertions of the values 11, 32, 20 in the tree in figure 3.

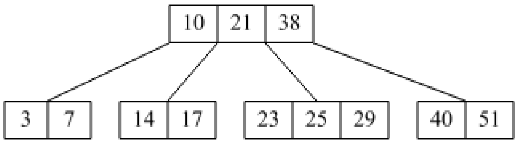

Figure 3: B-tree `B1` for insertion, degree 2

2. Draw the tree resulting from the deletion of the value 15 in the tree in figure 4.

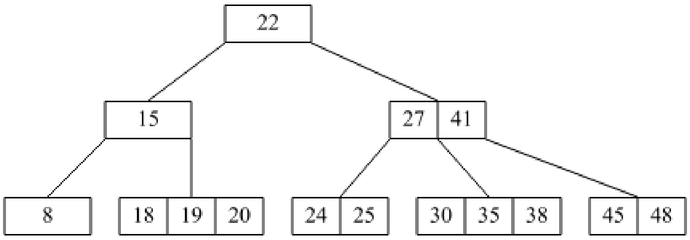

Figure 4: B-tree `B2` for deletion, degree 2

Exercise 5 (Mystery – *4 points*)

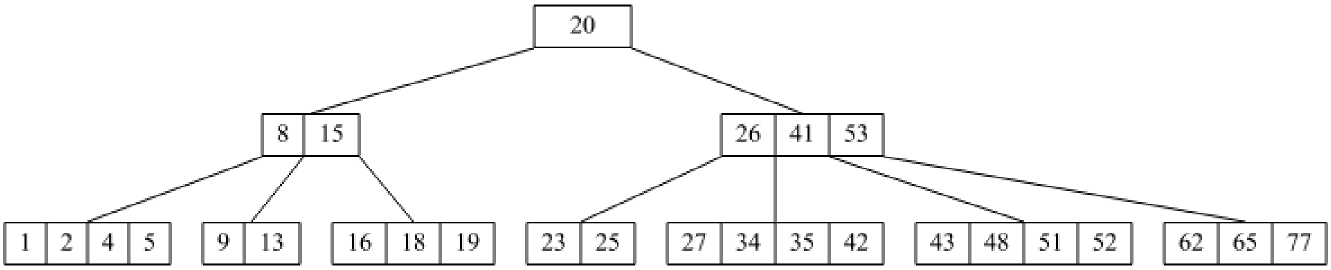

Figure 5: Tree `B3`

Let `mystery` be defined below:

```
def mystery(B, a, b):
    if B.keys[0] < a or B.keys[B.nbkeys-1] > b:
        return False
    else:
        for i in range(B.nbkeys-1):
            if B.keys[i] >= B.keys[i+1]:
                return False
        if B.children == []:
            return True
        else:
            i = 0
            while i < B.nbkeys and mystery(B.children[i], a, B.keys[i]):
                a = B.keys[i]
                i += 1
            return i == B.nbkeys and mystery(B.children[B.nbkeys], a, b)
```

1. For each of the following calls:
   - what is the returned result?
   - how many calls to `mystery` have been done?

   (a) `mystery(B2, 0, 92)` with `B2` the tree in figure 4
   (b) `mystery(B3, 0, 20)` with `B3` the tree in figure 5
   (c) `mystery(B3, 1, 99)` with `B3` the tree in figure 5

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

Appendix

Trees

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

***First child - right sibling* implementation**

- `B`: classe `TreeAsBin`
- `B.key`
- `B.child` : le premier fils
- `B.sibling` : le frère droit

B-Trees

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

- The empty tree is `None`
- The non empty tree is an object of the class `BTree` assumed imported.
  - `B.degree` is the degree (the order) of the B-Trees we deal with: it is a given constant!
  - `B.keys`: key list
  - `B.nbkeys` = `len(B.keys)`
  - `B.children`: child list (`[]` for leaves)
  - `B = BTree(key_list, child_list)` build a new tree

Other authorised functions and methods

As usual: `len`, `range`, `min, max, abs`.

Also, on lists:

```
>>> help(list.insert)
...     L.insert(index, object) -- insert object before index

>>> help(list.pop)
...     L.pop([index]) -> item -- remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

>>> help(list.append)
...     L.append(object) -> None -- append object to end
```

Your functions

You can write your own functions as long as they are documented: give their specifications (we must know what they do).

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