# Algorithmics
# Midterm #3 (C3)

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

*10 mars 2020 - 14h45*

---

## 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.

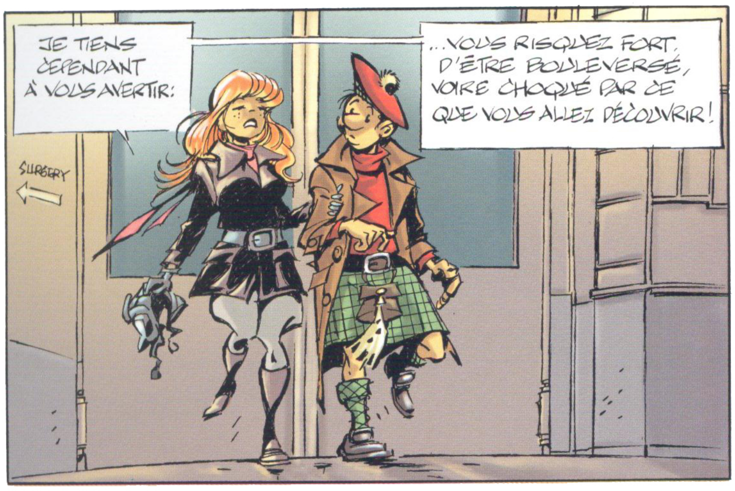
□ The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.

□ **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

---


JE TIENS CEPENDANT À VOUS AVERTIR:
...VOUS RISQUEZ FORT D'ÊTRE BOULEVERSÉ, VOIRE CHOQUÉ PAR CE QUE VOUS ALLEZ DÉCOUVRIR!
SURGERY

**Exercise 1 (Hashing – *2 points*)**

Assume the following set of key $E$={beck, cale, clapton, hendrix, hooker, king, richards, vaughan, winter, young} and the table 1 of hash values associated with each key of this set $E$. These values are lying between 0 and 10 ($m = 11$).

| | |
|---|---|
| beck | 10 |
| cale | 10 |
| clapton | 4 |
| hendrix | 5 |
| hooker | 6 |
| king | 8 |
| richards | 3 |
| vaughan | 8 |
| winter | 1 |
| young | 5 |

Table 1: Hash values

Present the collision resolution for adding all the keys of the set $E$ in the order of the table 1 (from `beck` to `young`):

1. using the linear probing principle with an offset coefficient $d = 3$.
2. using the hashing with separate chaining principle.

**Exercise 2 (Draw to win – *2 points*)**

Let `G` be the digraph `<S, A>` with:

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

1. Give the graphical representation of the digraph `G`.
2. Fill-in the vector `indegree` such that $\forall\ i \in [1, Card(S)]$, `indegree[i]` is the indegree of `i` in `G`.

**Exercise 3 (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.

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

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.

**Exercise 5 (B-trees: Minimum deletion – *6 points*)**

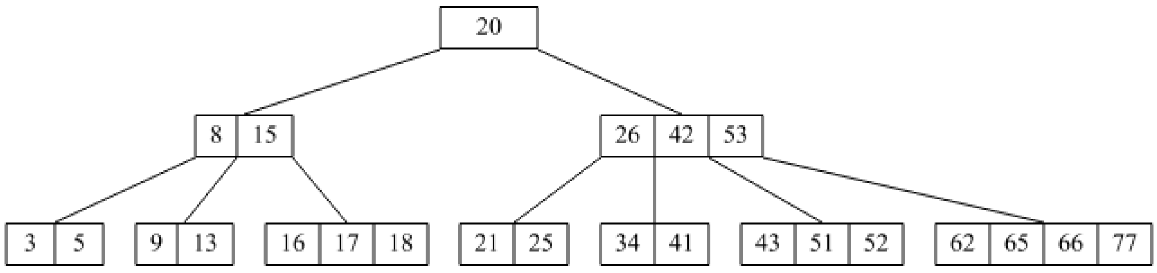


Figure 1: B-arbre

1. The tree in figure 1 is a Btree. What is the order (minimum degree) of this tree?
2. Using the "in going down" principle, draw the tree resulting from the deletion of the value 3.
3. Write the recursive function `__delmin(B)` that deletes the minimum value in the nonempty B-tree $B$. The function returns the deleted value.
   - ▷ Use the precautionary principle (in going down).
   - ▷ Use the functions given in appendix!
   - ▷ The function `__delmin` will be called by the following function that returns the tree and the deleted value:

```python
def del_min(B):
    """
    delete the minimum value of a BTree
    return (the tree and the value deleted, None if no minimum)
    """
    x = None
    if B:
        x = __delmin(B)
        if B.nbkeys == 0:
            if B.children:
                B = B.children[0]
            else:
                B = None
    return (B, x)
```

# Appendix

## Trees

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

**Classical implementation**

- T: classe `Tree`
- `T.key`
- `T.children` : listes des fils (`[]` pour les feuilles)
- `T.nbchildren` = `len(T.children)`

***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.

**Reminder:**

- 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` : liste des clés
- `B.nbkeys` = `len(B.keys)`
- `B.children` : liste des fils (`[]` pour les feuilles)

### Fonctions données

$t$ est le degré des B-arbres.

- The function `leftRotation` $(B, i)$ makes a rotation form child $i+1$ to child $i$.
  Conditions: the tree $B$ exists, its child $i$ exists and its root is not a $2t$-node, its child $i+1$ exists and its root is not a $t$-node.

- The function `rightRotation` $(B, i)$ makes a rotation form child $i-1$ to child $i$.
  Conditions: the tree $B$ exists, its child $i$ exists and its root is not a $2t$-node, its child $i-1$ exists and its root is not a $t$-node.

- The function `merge` $(B, i)$ merge $B$ children $i$ and $i+1$.
  Conditions: the tree $B$ exists and its root is not a $t$-node, children $i$ and $i+1$ exist and their roots are $t$-nodes.

## Other authorised functions and methods

**As usual:** `len`, `range`.

**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 (we have to know what they do).
In any case, the last written function should be the one which answers the question.