Algorithmique
Contrôle n° 3 (C3)

Info-spé (S3)
Epita

7 novembre 2017 - 14 : 45

Consignes (à lire) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - — Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - — Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - — Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - — Aucune réponse au crayon de papier ne sera corrigée.
- ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- ☐ **Le code :**
  - — Tout code doit être écrit dans le langage `Python` (pas de C, Caml, Algo ou autre).
  - — **Tout code `Python` non indenté ne sera pas corrigé.**
  - — Tout ce dont vous avez besoin (classes, fonctions, méthodes) est indiqué dans l'énoncé !
  - — Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).
    Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.
- ☐ Durée : 2h00

Du hachage

Exercice 1 (Quelques résultats différents – *6 points*)

1. Considérant que les collisions sont résolues par hachage coalescent, représenter la table de hachage de 11 éléments correspondant à l'application de la fonction de hachage

$$h(x) = (2x + 5)\ mod\ 11,$$

sur les clés $12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5$.

2. En reprenant les éléments de la question précédente et en considérant que les collisions sont résolues par hachage linéaire, représenter la table de hachage de 11 éléments qui en résulte.

3. En reprenant les éléments de la première question et en considérant que les collisions sont résolues par double hachage à l'aide de la deuxième fonction de hachage

$$d(x) = 7 - (x\ mod\ 7)$$

qui combinée avec $h(x)$ donne la table 1 d'essais successifs associés à chaque clé, représenter la table de hachage de 11 éléments qui en résulte.

TABLE 1 – valeurs d'essais successifs

Elt	h(x)	2	3	4	5	6	7	8	9	10	11
12	7	9	0	2	4	6	8	10	1	3	5
44	5	10	4	9	3	8	2	7	1	6	0
13	9	10	0	1	2	3	4	5	6	7	8
88	5	8	0	3	6	9	1	4	7	10	2
23	7	1	6	0	5	10	4	9	3	8	2
94	6	10	3	7	0	4	8	1	5	9	2
11	5	8	0	3	6	9	1	4	7	10	2
39	6	9	1	4	7	10	2	5	8	0	3
20	1	2	3	4	5	6	7	8	9	10	0
16	4	9	3	8	2	7	1	6	0	5	10
5	4	6	8	10	1	3	5	7	9	0	2

Des arbres

Exercice 2 (Préfixe - Suffixe – *3 points*)

Dans cet exercice, on se propose de remplir un vecteur avec les clés d'un arbre général. On place chaque clé **deux** fois dans le vecteur : lors de la première rencontre (ordre préfixe) et lors de la *dernière* rencontre (ordre suffixe) du parcours profondeur.

Écrire la fonction `prefsuff(T)` qui, à partir de l'arbre `T`, construit et retourne le vecteur de clés (représenté par une liste en Python) en respectant l'ordre décrit avec l'implémentation *premier-fils-frère droit* (voir l'exemple ci-dessous).

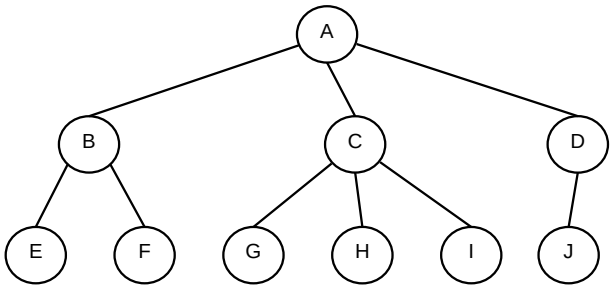

FIGURE 1 – Arbre général T_2

Voici comment sera rempli le tableau après parcours profondeur de l'arbre de la figure 1 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	B	E	E	F	F	B	C	G	G	H	H	I	I	C	D	J	J	D	A

Exercice 3 (B-tree or not B-tree... – *5 points*)

On désire vérifier si un B-arbre est bien "ordonné", c'est-à-dire si la relation d'ordre est bien respectée partout. Pour cela vous devez écrire la fonction récursive `test_Btree` qui prendra en paramètres le B-arbre à tester, ainsi que 2 valeurs qui représentent les bornes de l'intervalle sur lequel on travaille.

L'appel de votre fonction sera fait de la manière suivante :

```
def test_Btree_ordered(B):
  return (B == None) or test_Btree(B, −∞, +∞)
```

Avec $-\infty$ et $+\infty$ les valeurs extrêmes du type des éléments de l'arbre

---

Exercice 4 (B-Arbres : insertions – *4 points*)

Compléter la fonction récursive `__insert(`B`, `x`)` qui insère x dans le B-arbre non-vide B sauf x est déjà présente.

- ▷ Utiliser le principe de précaution (à la descente).
- ▷ Utiliser la procédure d'éclatement dont les spécifications sont les suivantes :
  La procédure `split(`B`, `i`)` éclate le fils n°i de l'arbre B.
  — L'arbre B existe (est non vide) et sa racine n'est pas un $2t$-nœud.
  — Le fils i de B existe et sa racine est un $2t$-nœud.
- ▷ On suppose implémentée la fonction `search_pos` dont les spécifications sont les suivantes :
  La fonction `search_pos(`L`, `x`)` cherche la valeur x dans la liste L non vide. Elle retourne la position de x dans L s'il est présent, sa position "virtuelle" dans le cas contraire.
- ▷ La fonction `__insert` sera appelée par la fonction suivante :

```
1    def insert(B, x):
2        if B ==  None:
3            return BTree([x])
4        else:
5            if B.nbkeys == 2 * B.degree - 1:     # root split
6                B = BTree([], [B])
7                split(B, 0)
8            __insert(B, x)
9            return B
```

Exercice 5 (B-arbres et mystère – *2 points*)

```
1    def build(nodes, degree):
2        BTree.degree = degree
3        B = BTree(nodes[0])
4        end = B.nbkeys + 2
5        q = Queue()
6        q.enqueue((B, 1))
7        while not q.isempty():
8            (cur, pos) = q.dequeue()
9            if pos < len(nodes):
10               for i in range(pos, pos + cur.nbkeys + 1):
11                   new = BTree(nodes[i])
12                   cur.children.append(new)
13                   q.enqueue((new, end))
14                   end += new.nbkeys + 1
15       return B
```

Quels paramètres ont été passés à la fonction `build` pour obtenir l'arbre de la figure 2 ?

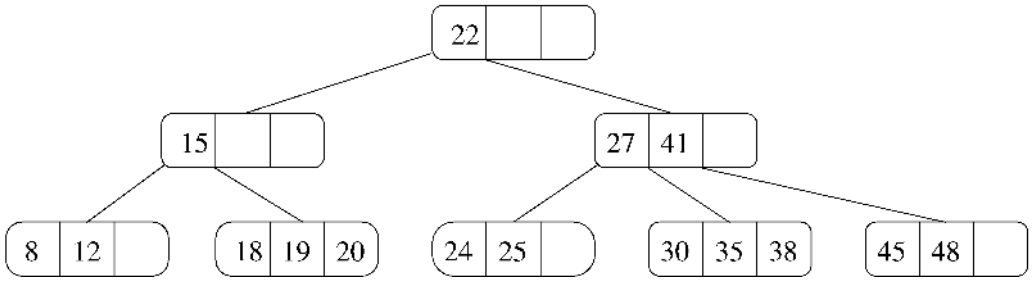

FIGURE 2 – B-tree

Annexes

Les arbres généraux

Les arbres (généraux) manipulés ici sont les mêmes qu'en td.

Implémentation *premier fils - frère droit*

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

B-Trees

Les B-arbres manipulés ici sont les mêmes qu'en td.

```
    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)
```

Fonctions et méthodes autorisées

Les fonctions que vous pouvez utiliser :

Comme d'habitude : `len`, `range`.

En plus, sur les listes :

```
>>> help(list.insert)
Help on method_descriptor:
insert(...)
    L.insert(index, object) -- insert object before index

>>> help(list.pop)
Help on method_descriptor:
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)
Help on method_descriptor:
append(...)
    L.append(object) -> None -- append object to end
```

Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.