Nom	
Prénom	
Groupe	

Note	

1	
2	
3	
4	
5	

Algorithmique
Arbres et Recherche II

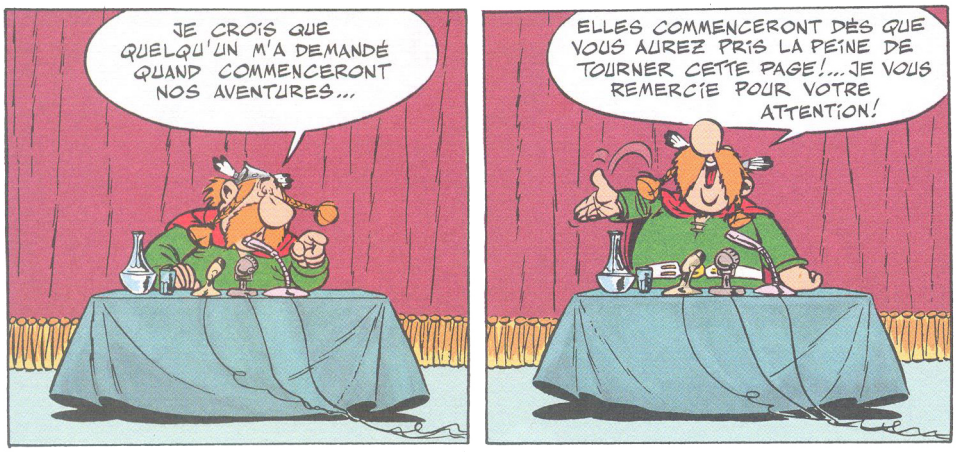
Spé S3 Epita

Examen B5

22 octobre 2024

Consignes (à lire) :

- ☐ Vous devez répondre directement **sur ce sujet**.
  - — Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées**.
  - — 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.
- ☐ **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.**
  - — Les seules classes, fonctions, méthodes que vous pouvez utiliser sont données en **annexe**.
  - — Vos fonctions doivent impérativement respecter les exemples d'applications donnés.
  - — 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.
  - — Comme d'habitude l'optimisation est notée. Si vous écrivez des fonctions non optimisées, vous serez notés sur moins de points. [1]
- ☐ Durée : 2h00



1. Des fois, il vaut mieux moins de points que pas de points.

Exercice 1 (Hachage coalescent – *3 points*)

On considère l'ensemble de clés données directement sous forme entière que l'on veut stocker dans une table de hachage de taille $m = 12$ (indicée de 0 à $m - 1$).

Soit la fonction de hachage : $h(x) = x \ mod \ m$.

1. Donner les valeurs de hachage associées aux éléments 15, 24, 125, 4, 26, 6, 78, 55, 89.
2. En considérant la fonction h et la gestion des collisions à l'aide du hachage coalescent, remplir la table de hachage obtenue après insertions successives des éléments suivants (dans cet ordre) : 15, 24, 125, 4, 26, 6, 78, 55, 89.

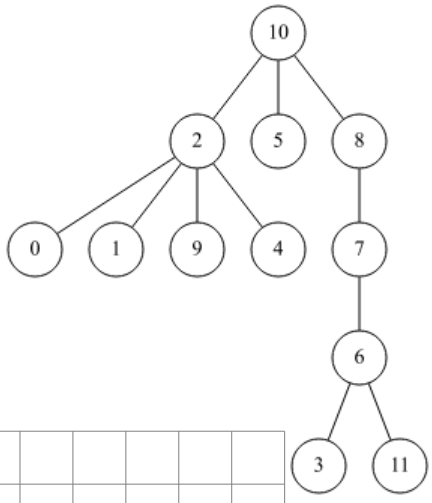
1. Valeurs de hachage :

elt	**valeur**
15	
24	
125	
4	
26	
6	
78	
55	
89	

2. Table de hachage :

	elt	lien
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Exercice 2 (Arbres : Profondeur moyenne – *5 points*)



En utilisant **obligatoirement un parcours profondeur**, écrire la fonction `internal_average_depth(T:Tree)` qui retourne la profondeur moyenne interne de l'arbre `T` en implémentation par n-uplets (`Tree`) si $taille(\texttt{T}) > 1$, 0 sinon.

Exemple d'application, avec `T` l'arbre de la figure ci-contre :

```
>>> internal_average_depth(T)
1.4 # 7/5
```

Exercice 3 (Trees : same arity – *6 points*)

Rappel : Dans un arbre général, l'*arité* d'un nœud est son nombre de fils.

Écrire la fonction `same_arity(B:TreeAsBin)` qui vérifie si tous les nœuds internes de l'arbre `B` en implémentation premier fils-frère droit (`TreeAsBin`) ont la même arité.

Exercice 4 (B-arbre : insertions et suppression – *3 points*)

Pour chaque question, utiliser le principe "à la descente" (principe de précaution) vu en td (hors bonus).

1. Dessiner l'arbre après insertions successives des valeurs 25, 3 et 41 dans l'arbre de la figure 1.

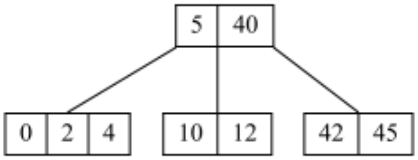


Figure 1 – B-arbre `B1` pour insertions, degré 2

2. Dessiner l'arbre après suppression de la valeur 50 dans l'arbre de la figure 2.

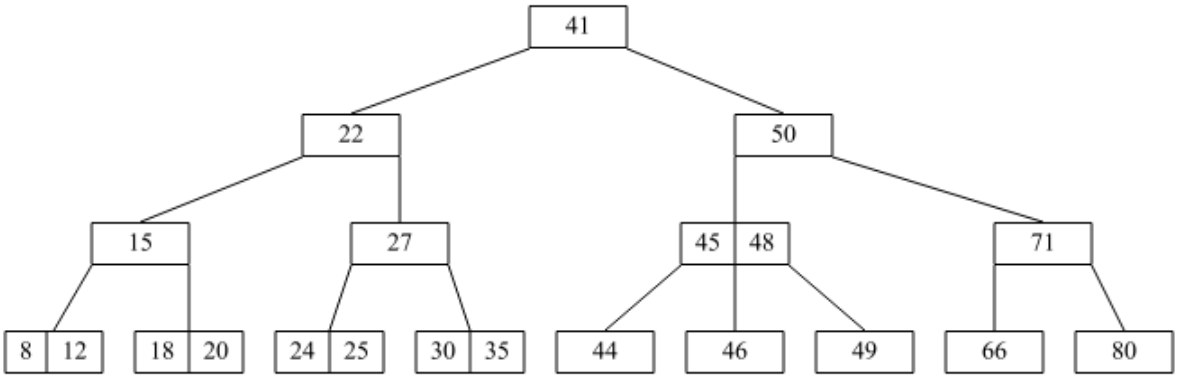


Figure 2 – B-arbre `B2` pour suppression, degré 2

Exercice 5 (What ? – *3 points*)

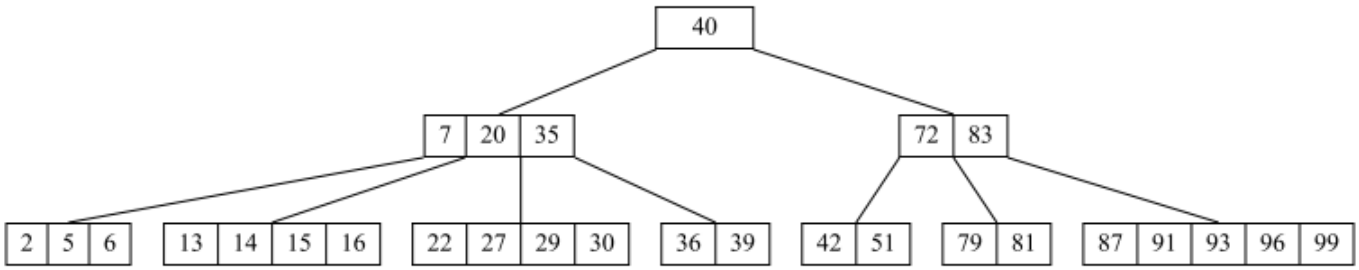


Figure 3 – B-arbre `BT`

Soit la fonction `what` définie ci-dessous :

```
def __aux(B, x, p):
    i = binary_search_pos(B.keys, x)
    f = i < B.nbkeys and B.keys[i] == x
    if B.children == []:
        if f:
            if i == 0:
                return p
            else:
                return B.keys[i-1]
        else:
            return None
    else:
        if f:
            B = B.children[i]
            while B.children != []:
                B = B.children[B.nbkeys]
            return B.keys[B.nbkeys-1]
        else:
            if i != 0:
                p = B.keys[i-1]
            return __aux(B.children[i], x, p)

def what(B, x):
    if B != None:
        return __aux(B, x, None)
    else:
        return None
```

Soit `BT` l'arbre de la figure 3. Quel sera le résultat de chacune des applications suivantes ?

`what(BT, 2)`	`what(BT, 20)`	`what(BT, 40)`	`what(BT, 41)`	`what(BT, 42)`	`what(BT, 87)`

Annexes

Les arbres généraux

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

Implémentation classique

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

Implémentation *premier fils - frère droit*

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

B-Trees

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

- L'arbre vide est `None`
- L'arbre non vide est un objet de la class `BTree`, que l'on suppose importée
  - `B.degree` est le degré (l'ordre) des B-arbres que l'on manipule : c'est une constante donnée !
  - `B.keys` : liste des clés
  - `B.nbkeys` = `len(B.keys)`
  - `B.children` : liste des fils (`[]` pour les feuilles)

Fonction donnée

- `binary_search_pos(`L`, `x`)` qui retourne la position de x dans la liste triée en ordre croissant L, ou la position où il devrait être s'il n'est pas présent.

Autres fonctions et méthodes autorisées

Comme d'habitude : `len`, `range`, `min, max, abs`

Les méthodes de la classe `Queue`, que l'on suppose importée :

- `Queue()` retourne une nouvelle file ;
- q`.enqueue(`e`)` enfile e dans q ;
- q`.dequeue()` supprime et retourne le premier élément de q ;
- q`.isempty()` teste si q est vide.

Vos fonctions

Vous pouvez également écrire vos propres fonctions à condition qu'elles soient documentées : **donnez leurs spécifications** (on doit savoir ce qu'elles font, la signification des paramètres).

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