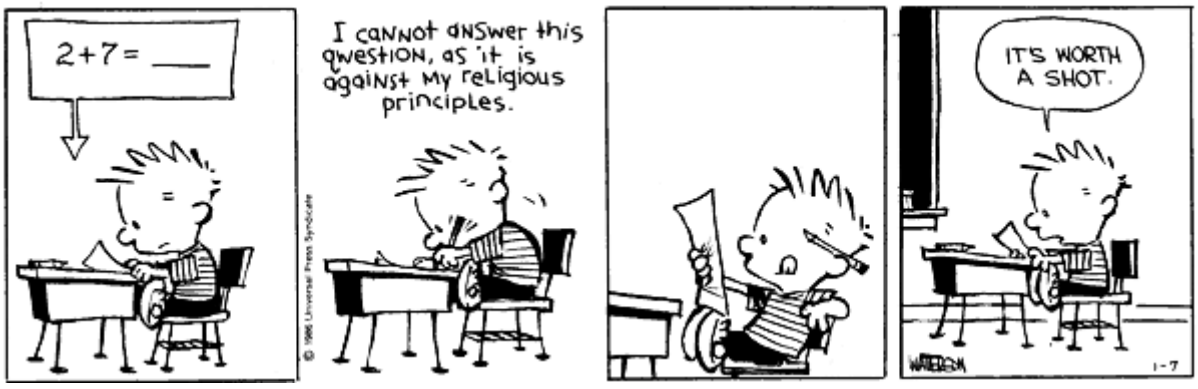
Algorithmique
Contrôle n° 3

Info-spé S3#
Epita

6 avril - 10h

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 en **annexes** !
  - — 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 questions – *4 points*)

1. Citer 3 méthodes de hachage de base.
2. Citer une méthode de hachage utilisant une fonction d'essais successifs.
3. Quelle méthode de résolution des collisions ne nécessite pas un tableau de hachage de taille supérieure ou égale au nombre de clés à hacher ?
4. Quelle type de recherche est incompatible avec le hachage ?

Des arbres

Exercice 2 (Sérialisation – *4 points*)

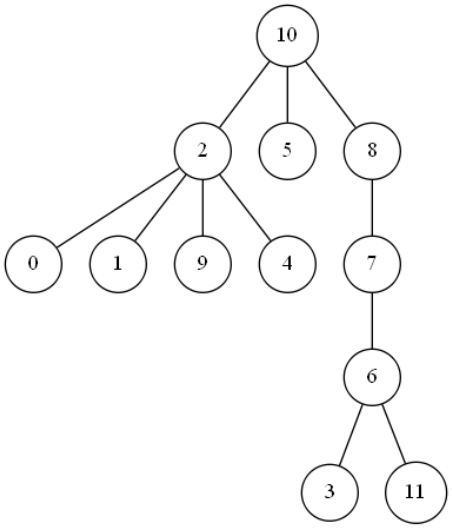


FIGURE 1 – Arbre général T

Nous allons nous intéresser à une représentation alternative des arbres généraux : les vecteurs de pères. Cette représentation est linéaire et peut donc être utilisée pour stocker notre arbre dans un fichier (sérialisation).

Le principe est simple : à chaque nœud de l'arbre on associe un identifiant unique (ici la clé contenue dans chaque nœud), sous la forme d'un entier compris entre 0 et la taille de l'arbre - 1. On construit ensuite un vecteur où la case i contient l'identifiant du père du nœud d'identifiant i. La racine de l'arbre aura pour père -1.

1. Donner le vecteur de pères pour l'arbre de la figure 1
2. Écrire la fonction `buildParentVect(`T`,` n`)` qui retourne le vecteur (représenté par une liste en Python) de pères correspondant à l'arbre T (implémentation "premier fils - frère droit") de taille n.

Exercice 3 (Représentation par listes – *5 points*)

Soit un arbre général A défini par $A =< o, A_1, A_2, ..., A_N >$. Nous appellerons *liste* la représentation linéaire suivante de A : (o A_1 A_2 ... A_N).

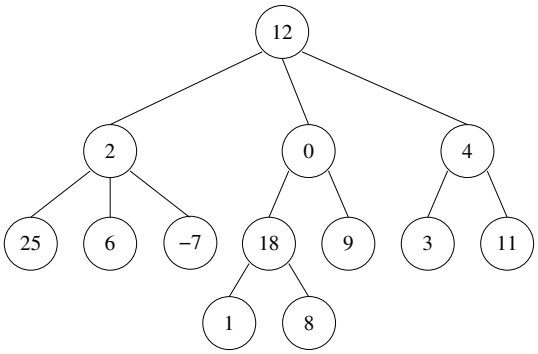


FIGURE 2 – Arbre correspondant à la *liste* "(12(2(25)(6)(-7))(0(18(1)(8))(9))(4(3)(11)))"

On souhaite construire un arbre général (implémentation "classique") à partir d'une *liste* (une chaîne de caractères s). Compléter la fonction `__fromList(`s`, `i`)` donnée.

Les clés de l'arbre résultat sont du type `int`.

Des B-arbres

Exercice 4 (B-arbre : suppression – *2 points*)

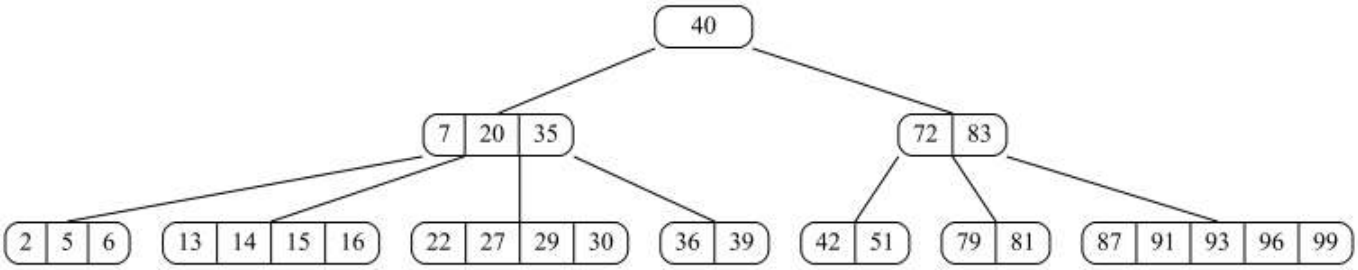


FIGURE 3 – B-tree

1. L'arbre de la figure 3 est un B-arbre. De quel ordre (degré minimum) est-il ?
2. En utilisant le principe "à la descente", dessiner l'arbre après suppression de la valeur 72.

Exercice 5 (Au suivant – *5 points*)

Écrire la fonction `findNext(B, x)` qui retourne la clé de `B` immédiatement supérieure à `x`. La fonction renvoie `None` si une telle valeur n'existe pas.

Dans l'arbre de la figure 3 :

— `findNext(B, 5)` retourne `6`

— `findNext(B, 6)` retourne `7`

— `findNext(B, 7)` retourne `13`

— `findNext(B, 40)` retourne `42`

— `findNext(B, 41)` retourne `42`

— `findNext(B, 99)` retourne `None`

Annexes

Arbres généraux

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

Implémentation classique

```
    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)
```

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 :

— `len` sur les listes

— `append` sur les listes

— `range`

— `int` transforme son paramètre en entier.

— `min` et `max`, mais uniquement avec deux valeurs entières !

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.