Algorithmique et programmation I (CS101)

Cahier d'exercices.

Table des matières

TD 1
Correction d'algorithmes élémentaires

Objectifs

— Pratiquer la description d'algorithmes simples : entiers, tableaux.

— Première initiation à la preuve de correction.

1.1 Exponentiation

cf l'examen de CS101 de 2022-23.

1.2 Parcours de tableaux

EXERCICE #1 ▸ **Maximum d'un tableau**
Rappeler rapidement l'algorithme calculant le maximum d'un tableau.

— Trouver un invariant de la boucle for et prouver cet invariant.

— Terminer la preuve de correction.

EXERCICE #2 ▸ **Recherche d'un élément dans un tableau**
Prouver la correction de la recherche d'un élément dans un tableau avec rupture prématurée du flot, que l'on a réalisée en cours :

```
bool recherche1(int t[N], int el){
  for (int i=0; i<N; i++){
    if (t[i]==el) return true;
  }
  return false;
}
```

EXERCICE #3 ▸ **Recherche dans un tableau trié**
Écrire un algorithmique (itératif) qui recherche dans un tableau supposé trié, par dichotomie :

```
bool recherche_dicho2(int t[N], int e){
// Précondition N>1, et pour tout i vérifiant 0 <= i <= N-2, on a t[i]<=t[i+1]
```

Ensuite, trouver un invariant de la boucle while (ie suffisant pour prouver la fonction), et le prouver.

TD 2

Tris de tableaux (d'entiers)

Objectifs

- Réviser les tris de tableaux, pratiquer la preuve d'invariants
- Savoir résoudre des complexités d'algorithmes récursifs.
- Connaître un résultat important sur les tris par comparaison.

2.1 Pseudo-code des tris supposément connus

On rappelera la spécification commune de ces algorithmes de tris.

Algorithme 1 : Tri par sélection

```
Action tri-sélection(T)
    D/R : T : Tableau[N] d'entiers
    L : ideb,i : Entiers
    L : imin : Entier                                  {Indice de l'élément minimum courant}
    Pour ideb de 0 à N-2 Faire
        imin := ideb                                   {Recherche de l'indice de l'élément minimum}
        Pour i de ideb + 1 à N-1 Faire
            Si T[i] < T[imin] alors
                imin := i
            Fsi
        Fpour
        permuter(T,ideb,imin)
    Fpour
FAction
```

Algorithme 2 : Tri par insertion

```
Action tri-insertion(T)
    D/R : T : Tableau[N] d'entiers
    L : élt : Entier                                   {Valeur à insérer}
    L : ins : Entier                                   {Indice d'insertion de élt}
    Pour i de 1 à N-1 Faire
        élt := T[i]                                    {Initialisation}
        ins := i
        Tq (ins ≥ 1 et T[ins-1] > élt) faire
            T[ins] := T[ins-1];
            ins := ins - 1
        Ftq
        T[ins] := élt                                  {Insertion de T[i]}
    Fpour
FAction
```

Algorithme 3 : Tri fusion - fonction auxiliaire, récursive.

```
Action tri-fusion-bis(T,premier,dernier)
    D : premier,dernier : Entiers
    D : T : Tableau[N] d'entiers
    L : milieu : entier
    Si premier<dernier alors
        milieu ← (premier+dernier)/2 ;
        tri-fusion-bis(T,premier,milieu);
        tri-fusion-bis(T,milieu+1,dernier) ;
        fusion(T,premier,milieu,dernier)
    Fsi
FAction
```

Algorithme 4 : Tri fusion - “toplevel”

```
Action tri-fusion(T)
    D : T : Tableau[N] d'entiers
    Si N>1 alors
        tri-fusion-bis(T,0,N-1)
    Fsi
FAction
```

Algorithme 5 : Algorithme de fusion, ne se comprend pas sans dessin

```
Action fusion(T,premier1,dernier1,dernier2)
    D : premier1,dernier1,dernier2 : Entiers
    D : T : Tableau[N] d'entiers
    L : fus : Tableau[dernier2-premier1+1] d'entiers
    L : c1,c2,premier2 : Entiers
    premier2 := dernier1+1
    c2 := premier2
    c1 := premier1
    Pour i de 0 à dernier2-premier1 Faire
        Si (c1≤dernier1 et (t[c1]<t[c2] ou c2> dernier2)) alors
            fus[i]← t[c1]
            c1++
        Sinon
            fus[i]← t[c2]
            c2++
        Fsi
    Fpour
    Pour i de 0 à dernier2-premier1 Faire
        t[premier1+i] ← fus[i]
    Fpour
FAction
```

EXERCICE #1 ▸ **Preuves de correction - à préparer**

Pour chaque tri quadratique :

— Exhiber et prouver un invariant de boucle bien choisi (et non trivial).

— Terminer la preuve de correction fonctionnelle.

Pour le tri-fusion :

— Donner une spécification de la fonction `fusion`.

— Faire une preuve de correction du tri-fusion, en supposant la fonction `fusion` correcte.

2.2 Complexité algorithmique des tris “comparaison”

On montre dans cette partie :

THÉORÈME 3. *La complexité en nombre de comparaisons du tri fusion est égale à $O(n \ln(n))$ avec n la taille du tableau à trier. Un tri par comparaisons ne peut pas avoir une complexité inférieure à $O(n \ln(n))$ comparaisons, le tri-fusion est donc* **optimal**.

EXERCICE #1 ▸ **Tri-fusion**
Pour le tri-fusion,

— En appelant $C(n)$ le nombre de comparaisons effectuées par l'algorithme (sur un tableau de taille n, montrer que $C(n) = \begin{cases} 2C(\frac{n}{2}) + n & \text{si } n \geq 2 \\ 1 & \text{sinon} \end{cases}$

— Conclure sur la complexité de CE tri sur les tabeaux. On résoudra la récurrence “à la main” en supposant que n est une puissance de 2.

— Est-ce que ce résultat est aussi valable sur des listes ?

EXERCICE #2 ▸ **Arbre de décision et complexité minimale**
Source : https ://zanotti.univ-tln.fr/ALGO/I31/BorneTrisComparatifs.html. *A consulter pour les justifications de l'usage des permutations pour l'analyse des tris par comparaison.*

On modélise l'ensemble des conditions rencontrées par un algorithme ainsi que les décisions qui en découlent par un arbre enraciné appelé arbre de décision. Chaque condition est représentée par un nœud auquel on associe autant de branches qu'il y a de décisions possibles. La racine de l'arbre contient la première condition rencontrée. Les noeuds internes montrent les comparaisons, et les feuilles possèdent une instance de permutation.

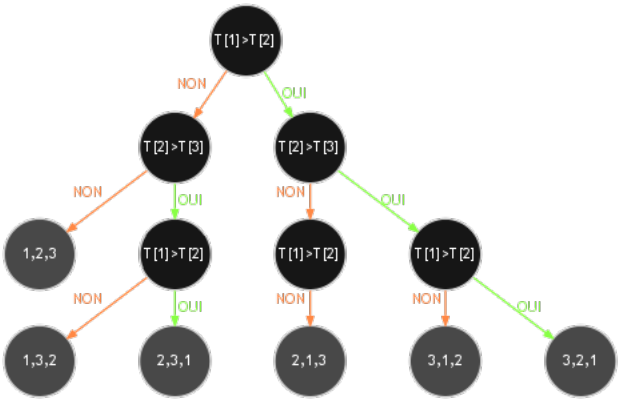


FIGURE 3.1 – Arbre de décision du tri-bulle pour les permutations de taille 3. Même source

— Construisez l'arbre de décision du tri par selection pour $n = 3$.
On admet que l'arbre de décision d'un algorithme de tri comparatif appliqué à des permutations de$\{1 \dots n\}$ contient $n!$ feuilles.

— Quelle est la hauteur minimale d'un arbre de décision comportant $n!$ noeuds ? Conclure.

EXERCICE #3 ▸ **Un tri linéaire, mais quelle est cette magie ?**
Réaliser un tri linéaire de tableau d'entiers positifs, en sachant que la valeur maximale de ce tableau est 6. Généraliser cet algorithme dans le cas d'un tableau avec peu de valeurs différentes.

TD 4

Arbres binaires, Arbres Binaires de recherche

Objectifs

— Pratiquer les algorithmes d'arbre.

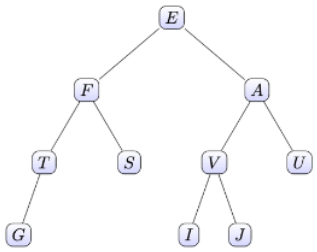
— Savoir calculer des complexités de ces algorithmes.

On commencera par (se) rappeler l'interface des arbres binaires du cours.

4.1 Échauffement : pratique des parcours

EXERCICE #1 ▸ **Parcours**

Donner les noms et les effets des différents parcours avec la fonction $F = print\_char$ sur l'arbre suivant :



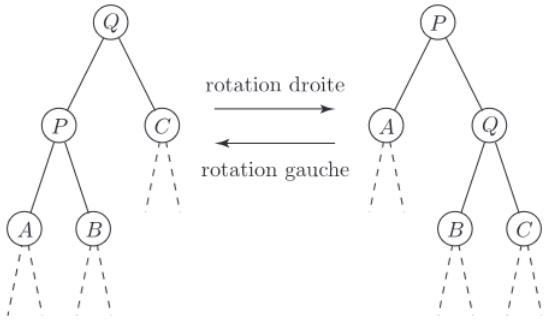
4.2 Arbres binaires de recherche

EXERCICE #2 ▸ **Définitions, tri**

Rappeler la définition d'un ABR, et prouver que le parcours infixe en profondeur se fait par valeurs croissantes des noeuds.

EXERCICE #3 ▸ **Rotations**

Une rotation est une opération qui transforme un arbre binaire de recherche en modifiant la position de certains nœuds tout en conservant l'ordre des éléments. Une rotation touche principalement deux nœuds, l'un étant le fils de l'autre et permet de faire remonter le fils tout en descendant le père pour inverser leur niveau dans l'arbre. Il existe deux rotations : droite et gauche, illustrées sur la figure suivante (source : V. Poupet, univ Montpellier). Remarquons que l'on ne modifie pas les sous-arbres enracinés en A,B,C.



Écrivez les fonctions rotation gauche et rotations droite et donnez leur complexité.

La suite de ce TD reprend une partie de l'épreuve disciplinaire du CAPES NSI de cette année. Les questions seront adaptées pendant la séance.

Épreuve disciplinaire

Préliminaires

Notes de programmation Python

Il est demandé d'accompagner les programmes Python d'explications de conception rédigées.

Vous disposez, pour répondre aux questions de ce sujet, des fonctions de listes suivantes :

— création d'une liste `li` de taille n dont tous les éléments ont pour valeur v : `li = [v]*n`.
— renvoi de la taille d'une liste `li` : `len(li)`.
— insertion (avec décalage à droite) à un indice i dans une liste `li` : `li.insert(i,v)`.
— ajout de la valeur v en fin de liste : `li.append(v)`.
— position du premier élément de la liste égal à v (en comptant à partir de 0) : `li.index(v)`.
— énumération des éléments d'une liste, dans l'ordre : `for x in li`.
— construction de listes par *compréhension* : `[x**2 for x in li]` est une nouvelle liste construite en mettant au carré chaque élément de la liste `li`.
— les fonctions permettant de sélectionner des tranches (*slices*) : `li[start:end]` sélectionne la sous-liste d'indices de `start` à `end-1`.

Il est autorisé d'utiliser l'évaluation gauche droite paresseuse des expressions booléennes, par exemple, si l'expression `expr_bool1` s'évalue à `False`, alors l'expression `expr_bool1 and f(42)` est directement évaluée à False (sans appel de la fonction `f`).

Enfin, on utilise la possibilité en Python des versions supérieures à 3.10 de décrire les types des paramètres et de retour de fonction, par exemple : `foo(a:int, b:int)->bool` est une fonction nommée `foo`, à deux paramètres entiers, retournant un Booléen.

Notes de complexité

La complexité d'un algorithme est le nombre d'opérations élémentaires réalisées par celui-ci et calculée en fonction des données d'entrées, et selon le contexte (à rappeler), exprimée dans le meilleur cas, le pire cas, ou en moyenne. Cette complexité sera dans tout ce sujet exprimée à l'aide de l'opérateur O, dont on rappelle la définition ici : on dit que $f = O(g)$ s'il existe une constante K et une constante M telle que pour tout $n > M$, $f(n) \leqslant Kg(n)$.

On rappelle que les *listes* Python sont en fait des tableaux redimensionnables. En conséquence, le calcul de la taille, l'accès à un élément et l'ajout en fin de liste sont considérés comme des opérations élémentaires, avec une complexité $O(1)$. L'insertion et le parcours sont des opérations linéaires en la taille de la liste.

Vocabulaire d'arbres

Dans ce sujet :

— les arbres sont des graphes acycliques connectés, enracinés et étiquetés ;
— un nœud sans enfant est dénommé feuille ;
— on appelle la longueur d'un chemin le nombre de nœuds de ce chemin, extrémités inclues ;
— on appelle arité d'un nœud le nombre d'enfants de ce nœud ; par extension on appelle arité d'un arbre le maximum de l'arité de ses nœuds. Un arbre binaire est d'arité 2.
— on appelle distance entre deux nœuds A et B la valeur $l - 1$, où l désigne la longueur du chemin (A, B) ;
— on appelle niveau d'un nœud d'un arbre la distance de ce nœud à la racine de l'arbre (la racine d'un arbre est de niveau 0) ;
— la hauteur d'un arbre est la longueur du plus grand chemin de la racine à une feuille (en nombre de nœuds) ;
— un arbre est équilibré si tous les chemins de la racine aux feuilles ont la même longueur ;

Rappel mathématique

$$\text{Pour } x \neq 1 \ : \ \sum_{i=0}^{i=n-1} x^i = \frac{1-x^n}{1-x}$$

1 Arbres Binaires de Recherche

Un arbre binaire de recherche est un *arbre binaire* (i.e. chaque nœud possède 0, 1 ou 2 enfants), dans lequel les nœuds sont étiquetés par des *entiers naturels* appelés *clefs*. De plus, si l'on considère un nœud de clef k, alors chaque nœud de son sous-arbre gauche (resp. droit) a une clef inférieure ou égale à k (resp. supérieure ou égale). Un exemple de tel arbre est donné à la Figure 1.

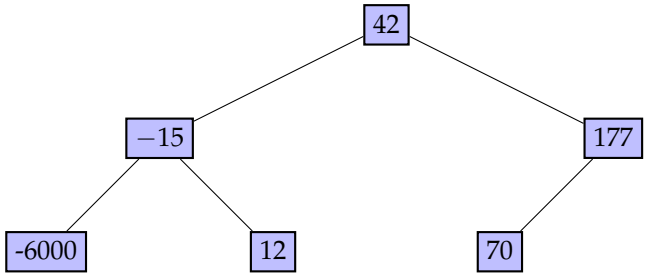


FIGURE 1 – Un arbre binaire de recherche

Nous proposons la définition suivante pour les arbres binaires en Python : tout d'abord, nous définissons un type `Noeud` sous forme d'une classe (un nœud a forcément deux enfants, qui sont soit des nœuds, soit `None`) :

```
class Noeud:
    """
    Noeud d'un arbre
    """
    def __init__(self, clef:int, gauche= None, droite= None):
        """ Initialisation, noeud non vide"""
        self.gauche = gauche
        self.droite = droite
        self.clef = clef
```

Par la suite, un arbre binaire est soit l'arbre vide (`None`) soit enraciné sur un nœud (`Noeud`).
L'arbre de la Figure 1 peut donc être construit de la façon suivante :

```
mon_arbre = Noeud(42, Noeud(-15, Noeud(-6000), Noeud(12)), Noeud(177, Noeud(70), None));
```

À ce stade, rien ne permet de s'assurer que l'arbre construit est un arbre binaire de recherche.

Question #1

Écrire une fonction récursive en Python de recherche d'une clef dans un arbre binaire de recherche (contenant au moins un nœud). Cette fonction aura la signature suivante :
`recherche(root : Noeud, k : int)->bool`

Question #2

Donner la complexité algorithmique de votre fonction en fonction du nombre de nœuds de l'arbre binaire de recherche. On distinguera des cas selon la forme des arbres en entrée.

Question #3

En utilisant un parcours récursif, écrire une fonction `collecte` qui collecte et ajoute dans une liste Python les valeurs des clefs des nœuds d'un arbre binaire de recherche par ordre croissant, et renvoie cette liste. On pourra utiliser la concaténation de listes et l'ajout en fin de liste (avec `append`) sans se préoccuper de la complexité.

Question #4

Nous proposons l'algorithme suivant pour déterminer si un arbre binaire est un arbre binaire de recherche :

```
    def est_ABR_1(self):
    """
    renvoie True ssi l'arbre est un ABR
    """
    temp = True
    val = self.clef
    if self.gauche is not None:
        temp = self.gauche.clef < val and self.gauche.est_ABR_1()
    if self.droite is not None:
        temp = temp and self.droite.clef > val and self.droite.est_ABR_1()
     return temp
```

En exhibant un arbre (non feuille) bien choisi, montrer que cet algorithme est incorrect.

Nous désirons maintenant proposer un algorithme correct : celui-ci est basé sur l'invariant structurel suivant : lorsque l'on observe un sous-arbre (d'un arbre binaire de recherche) enraciné en un nœud de clef k :

— Dans le sous-arbre de gauche, le maximum des clefs est inférieur ou égal à k.

— Dans le sous-arbre de droite, le minimum des clefs est supérieur ou égal à k.

Pour simplifier, il est supposé que l'on connaît une borne inférieure (`inf=INF`) et une borne supérieure (`sup=SUP`) des valeurs des clefs de l'arbre. L'algorithme consiste alors à propager un intervalle (initialement égal à `(inf,sup)`) de la racine vers les feuilles à chaque appel récursif du parcours.

Question #5

Recopier sur votre copie l'arbre de la Figure 1 en indiquant à côté de chaque nœud les intervalles propagés par l'algorithme décrit ci-dessus.

Question #6

Sur votre contre-exemple de la question 4, comment cet algorithme va-t-il décider que ce n'est pas un arbre binaire de recherche ?

Question #7

Implémenter la fonction `est_ABR_2_aux(self, inf:int, sup:int)->bool` qui propage les intervalles de haut en bas et décide si l'arbre binaire courant est un arbre binaire de recherche.

2 B-trees

Pour une première définition, nous nous reportons à Wikipédia :

> « En informatique, un arbre B (appelé aussi B-arbre par analogie au terme anglais « B-tree ») est une structure de données en arbre équilibré. Les arbres B sont principalement mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichiers. Ils stockent les données sous une forme triée et permettent une exécution des opérations d'insertion et de suppression en temps toujours logarithmique.
>
> Le principe est de permettre aux nœuds parents de posséder plus de deux nœuds enfants : c'est une généralisation de la notion d'arbre binaire de recherche. Ce principe minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage. De plus un B-tree grandit à partir de la racine, contrairement à un arbre binaire de recherche qui croît à partir des feuilles. »

Dans cette partie nous allons tout d'abord implémenter ces B-trees et en caractériser certaines opérations.

TD 5

B-trees

Objectifs

— Pratiquer les algorithmes d'arbre.

— Etude d'une variante des ABR.

Ce TD reprend une partie de l'épreuve disciplinaire du CAPES NSI de cette année. Les questions seront adaptées pendant la séance.

Pour une première définition, nous nous reportons à Wikipédia :

> « En informatique, un arbre B (appelé aussi B-arbre par analogie au terme anglais « B-tree ») est une structure de données en arbre équilibré. Les arbres B sont principalement mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichiers. Ils stockent les données sous une forme triée et permettent une exécution des opérations d'insertion et de suppression en temps toujours logarithmique.
>
> Le principe est de permettre aux nœuds parents de posséder plus de deux nœuds enfants : c'est une généralisation de la notion d'arbre binaire de recherche. Ce principe minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage. De plus un B-tree grandit à partir de la racine, contrairement à un arbre binaire de recherche qui croît à partir des feuilles. »

Nous allons étudier la structure de données, et implémenter un algorithme de recherche (sans montrer que la recherche est en moyenne en O(log(n) comparaisons avec n le nombre de noeuds).

2.1 Définition, algorithmes élémentaires

Contrairement à la section précédente, les B-trees ne seront pas binaires, mais d'arité quelconque. De plus, les nœuds ne contiennent pas une clef unique, mais un ensemble de clefs.

Définition
Un B-tree d'ordre t (cf. Figure 2) est un arbre qui satisfait les propriétés suivantes :

- (S1) Tous les chemins de la racine à une feuille ont la même longueur (le même nombre de nœuds), appelée hauteur de l'arbre et dénotée par h.
- (S2) La racine est une feuille ou bien a au moins 2 enfants.
- (S3) Chaque nœud qui n'est ni racine ni feuille possède au moins $t+1$ enfants.
- (S4) Chaque nœud a au plus $2t+1$ enfants.
- (C1) Les nœuds contiennent des clefs : la racine contient de 1 à $2t$ clefs ; et les autres nœuds, entre t et $2t$ clefs.
- (C2) Dans chaque nœud, les éléments sont stockés par ordre croissant.
- (C3) Chaque nœud qui contient ℓ clefs (et qui n'est pas une feuille) a $\ell+1$ enfants.
- (C4) Considérons un nœud P contenant ℓ clefs $x_0 \dots x_{\ell-1}$. Soient $P_0, \dots P_\ell$ ses enfants, et $K(P_i)_{0 \leqslant i \leqslant \ell}$ l'ensemble des clefs du sous-arbre dont la racine est P_i. Alors :
  - $\forall y \in K(P_0), y \leqslant x_0$;
  - $\forall 1 \leqslant i \leqslant \ell - 1, \forall y \in K(P_i),\ x_{i-1} \leqslant y \leqslant x_i$;
  - $\forall y \in K(P_\ell), x_{\ell-1} \leqslant y$;

Les propriétés sont de deux ordres : quatre propriétés **structurelles**, préfixées par « S », et des propriétés de contenu/de clefs, préfixées par « C ». Remarquons que la propriété C4 est une généralisation de la propriété fondamentale des arbres binaires de recherche.

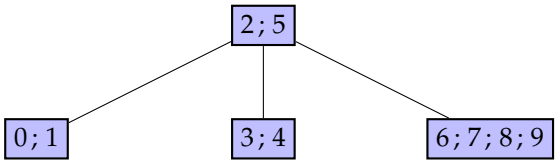
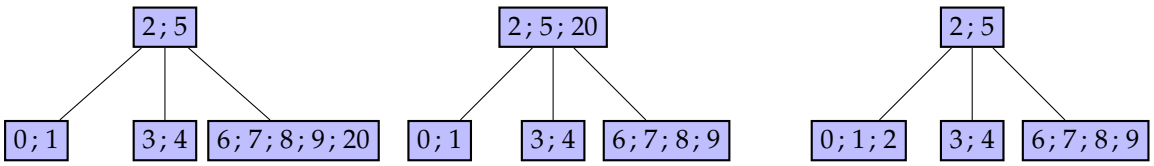


FIGURE 2 – Un exemple de B-tree d'ordre 2. Les nœuds contiennent un ensemble de clefs, triées par ordre croissant.

Question #8
Les arbres de la Figure 3 sont-ils des B-trees d'ordre 2 ?



(a) Arbre 1 (b) Arbre 2 (c) Arbre 3

FIGURE 3 – Quelques arbres.

L'implémentation du type B-tree en Python est fournie ci-dessous :

```
class BTreeNode:
  def __init__(self, t:int, f=False):
    self.feuille = f # True si le noeud est une feuille
    self.clefs = []   # Les valeurs des clefs du noeud
    self.enfants = []  # les enfants du noeud (qui sont des BTreeNode)
    self.ordre = t       # l'ordre du noeud
    self.n = 0       # nombre de clefs du noeud
```

```
class BTree:
  def __init__(self, t):
    self.root = BTreeNode(t,True)
    self.ordre = t  # l'ordre du btree
```

Ainsi l'appel `BTree(2)` renvoie une instance de la classe BTree d'ordre 2 représentant une feuille sans clef ni enfant. Remarquons en particulier que :

— Rien ne garantit les invariants de la définition d'un B-tree. Il faut donc bien faire attention à les maintenir.

— La numérotation des enfants et celle des clefs commencent à 0.

Par commodité, toutes les fonctions écrites seront des méthodes de la classe BTree.

Question #9

Écrire une fonction récursive d'impression d'un B-tree, `def print_node(self, niveau=0)`, qui affiche chacun des nœuds du `BTreeNode` courant, à partir du nœud x . Chaque nœud sera affiché sur une nouvelle ligne, avec le nombre de clefs qu'il contient, ainsi que les valeurs de ces clefs. Ainsi, sur l'exemple de la Figure 2, l'appel `B.root.print_node()` doit donner l'impression suivante :

```
Niveau  0   2:2 5
Niveau  1   2:0 1
Niveau  1   2:3 4
Niveau  1   4:6 7 8 9
```

2.2 Recherche dans un B-tree

La recherche d'une clef dans un B-tree va suivre le même motif que pour un arbre binaire de recherche : en effet, elle va combiner deux algorithmes : la recherche dans une liste triée (`keys`) et la recherche parmi les enfants (`child`) d'un nœud donné.

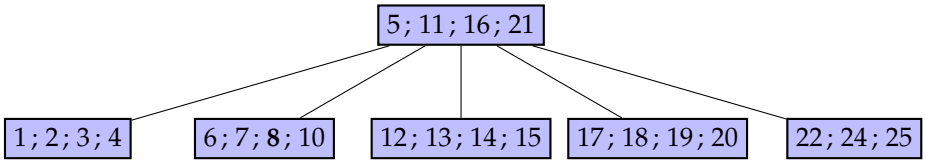


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

Question #10

Dans le B-tree de la figure 4, expliquer les étapes de recherche de la clef dont la valeur est 8. Utiliser les caractéristiques du B-tree pour ne pas réaliser un parcours entier de l'arbre.

Question #11

En déduire, pour un nœud fixé, une condition d'arrêt de la recherche dans ce nœud ainsi que le numéro du sous-arbre dans lequel rechercher récursivement.

Question #12

Implémenter en Python `recherche_clef(self : BTreeNode, k : int)-> bool` cherchant la clef k dans l'arbre courant à partir du nœud courant.

Étudions maintenant la complexité algorithmique de la recherche. Considérons des B-trees non vides, d'ordre $t > 1$, de hauteur $h \geqslant 1$. Dans un premier temps, nous allons compter le nombre de nœuds minimum N_{min} et maximum N_{max} d'un tel B-tree. Il est recommandé de procéder en comptant les nœuds par niveau.

Question #13

Montrer que $N_{min} = 1 + \frac{2}{t}\left((t+1)^{h-1} - 1\right)$ pour $h \geqslant 2$.

TD 6
Graphes

Sources
- https://www.irif.fr/~francoisl/DIVERS/l3algo1617-TD2.pdf
- https://www.irif.fr/~francoisl/DIVERS/l3algo-td5-1011.pdf
- http://www.gymomath.ch/javmath/polycopie/th_graphe4.pdf
- Feuille de TD L3 ENSL.
- Diverses Annales

6.1 Degré, connexité

EXERCICE #1 ▸ **Une preuve par construction**
Un graphe est dit k-régulier si tous ses sommets sont de degré k. Prouver la propriété suivante :

Pour tout entier n pair, $n > 2$, il existe un graphe 3-régulier composé de n sommets.

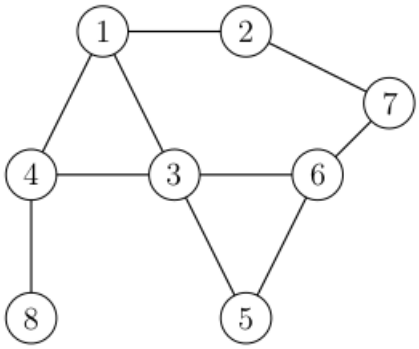
EXERCICE #2 ▸ **Degré**
Si m est le nombre d'arêtes d'un graphe G, montrer que

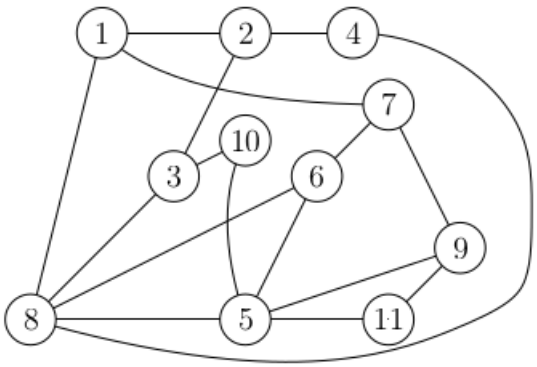
$$\sum_{v \in V(G)} d_G(v) = 2m.$$

EXERCICE #3 ▸ **Connexité**
Montrez que tout graphe connexe à n sommets a au moins n arêtes.

6.2 Parcours

EXERCICE #4 ▸ **Parcours en largeur**
Pour chacun des graphes suivants, donner l'ordre des noeuds rencontrés lors d'un parcours en largeur, en partant du sommet 1. Donner l'arbre résultant de ce parcours.





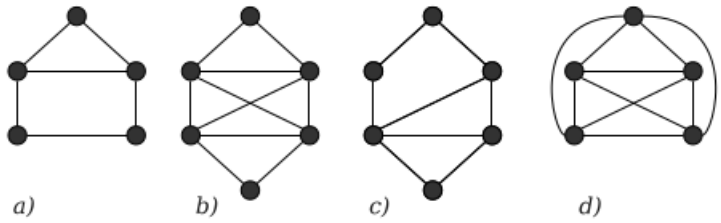
Quelle est la complexité du parcours en largeur avec les listes d'adjacence ?

EXERCICE #5 ▸ **Profondeur**
Donner l'ordre des noeuds visités dans le parcours en profondeur des deux graphes précédents à partir du noeud 1.

EXERCICE #6 ▸ **Applications des parcours**
Proposer un algorithme qui permet de déterminer si un graphe contient un cycle.

6.3 Cycles : graphes eulériens/hamiltoniens

EXERCICE #7 ▶ **Cycles Eulériens**
Un chemin est dit eulérien si il passe une et une seule fois par chacune des *arêtes* du graphe. Les graphes suivants possèdent-ils un cycle eulérien ?



EXERCICE #8 ▶ **CNS Chemin Eulérien**
Montrer qu'un graphe connexe admet un chemin eulérien ssi au plus 2 de ses sommets sont de degré impair.

EXERCICE #9 ▶ **Graphes Hamiltoniens**
Un cycle est dit Hamiltonien ssi il passe une et une seule fois par chaque sommet du graphe.
Les graphes suivants possèdent-ils un cycle hamiltonien ?

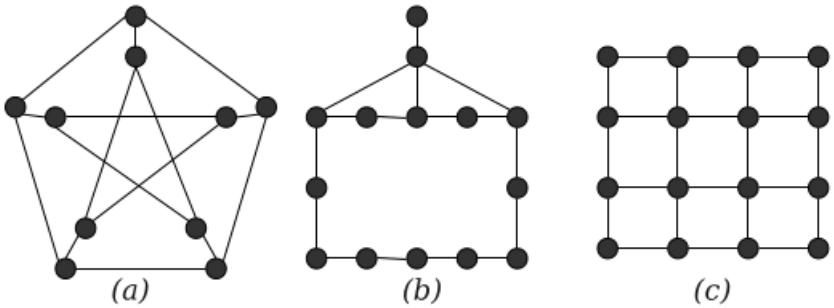


FIGURE 3 – (a) Graphe de Petersen 3-régulier, (b) maison agrandie, (c) grille

Donner un algorithme pour déterminer si un tel cycle existe.