

Nom : _____

Numéro de votre place : _____

Code permanent : _____

Directives pédagogiques :

- Inscrivez votre nom, prénom, code permanent et le numéro de votre place.
- Rendre visible votre carte d'identité étudiante.
- Lisez attentivement toutes les questions et **répondez directement sur le questionnaire.**
- Seule l'utilisation d'un crayon ou stylo est permise, **aucune documentation, calculatrice, téléphone cellulaire, ordinateur, ou autre objet permis.**
- Cet examen contient 5 questions pour 160 points
- Le barème est établi à environ 1 point par minute.
- Cet examen contient 22 pages, incluant 7 pages à la fin pour vos brouillons que vous pouvez détacher.
- **Écrivez lisiblement et détaillez vos réponses.**
- Vous avez 160 minutes pour compléter cet examen.

BONNE CHANCE et BON ÉTÉ !

| | |
|-------|-------|
| 1 | / 20 |
| 2 | / 20 |
| 3 | / 50 |
| 4 | / 40 |
| 5 | / 30 |
| Total | / 160 |

1. (20). L'algorithme suivant prend en entrée un tableau et renvoie le tableau avec tous les éléments dupliqués supprimés. Par exemple, si le tableau d'entrée est {1,3,3,2,4,2}, l'algorithme renvoie {1,3,2,4}. Quel est l'ordre de sa performance en temps d'exécution si l'ensemble S est implanté avec :

```
def setisize( E ):
    S = set()
    A = []
    for x in E:
        if not x in S:
            S.add( x )
            A.append( x )
    return A
```

- a) (10) Un arbre AVL ?

x in S se fait en temps dans $O(\log n)$, on le fait pour les n éléments de E , donc $O(n \log n)$.

S.add(x) se fait en temps dans $O(\log n)$, on le fait en pire cas pour les n éléments de E , donc $O(n \log n)$.

La somme des 2 opérations reste en temps dans $O(n \log n)$.

- b) (10) Une table de hachage ?

x in S se fait en temps dans $O(1)$, on le fait pour les n éléments de E , donc $O(n)$.

S.add(x) se fait en temps dans $O(1)$, on le fait en pire cas pour les n éléments de E , donc $O(n)$.

La somme des 2 opérations reste en temps dans $O(n)$.

2. (20) Supposez la table de hachage suivante, implémentée avec sondage linéaire et la fonction de hachage identité, $h(x) = x$.

| | | | | | | | | |
|---|----|---|----|---|----|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 18 | | 12 | 3 | 14 | 4 | 21 | |

- a) (10) Dans quel ordre les éléments ont-ils pu être ajoutés à la table ?
 ATTENTION : Il y a plusieurs bonnes réponses, et vous devez les donner toutes.
 Supposez que la taille de la table n'a jamais été modifiée et qu'aucun élément n'a été supprimé.

- A 9, 14, 4, 18, 12, 3, 21 (4 aurait dû être inséré à 4, donc non)
- B 12, 3, 14, 18, 4, 9, 21 (18 aurait dû être inséré à 0, donc non)
- C 12, 14, 3, 9, 4, 18, 21 (**OUI**)
- D 9, 12, 14, 3, 4, 21, 18 (**OUI**)
- E 12, 9, 18, 3, 14, 21, 4 (21 aurait dû être inséré à 6, donc non)

- b) (10) Sachant que lorsqu'on retire un élément de la table on peut libérer l'entrée qui était utilisée par cet élément ou mettre une valeur de disponibilité (e.g. AVAIL). Redessiner la table ci-haut après avoir retiré 3 et expliquez votre réponse.

| | | | | | | | | |
|---|----|---|----|-------|----|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 18 | | 12 | AVAIL | 14 | 4 | 21 | |

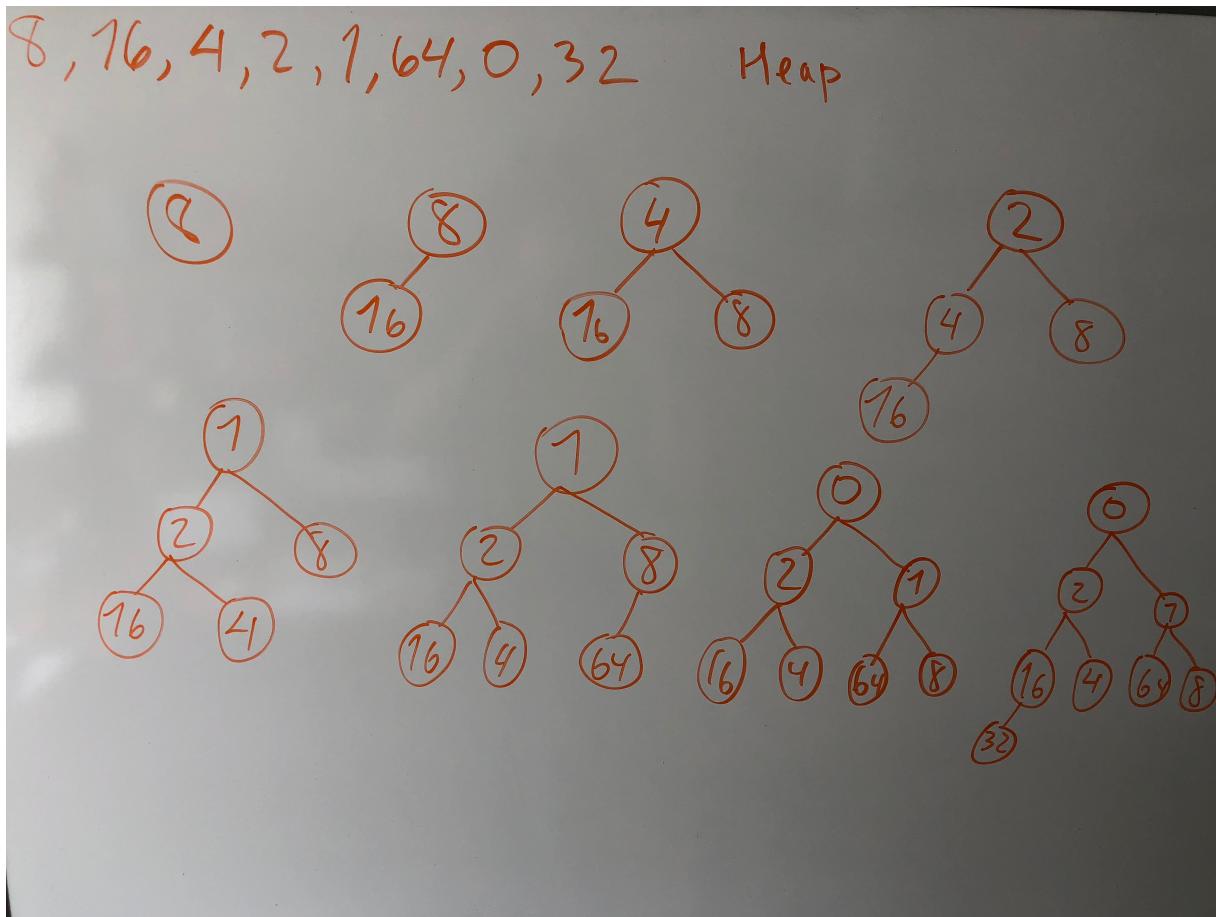
Il ne faut pas briser les regroupements. Ici, si on rend la case libre, on perd l'accès à la valeur 4, par exemple.

- c) (10) Si nous voulons une table de hachage qui stocke un ensemble de chaînes de caractères (String), une fonction de hachage possible est la longueur de la chaîne, $h(x) = \text{len}(x)$. Est-ce une bonne fonction de hachage ? Expliquez.

NON. Toutes les chaînes de caractères de la même longueur vont se retrouver avec le même code de hachage. La recherche dans ce cas peut dégénérer vers $O(n)$ plutôt que $O(1)$.

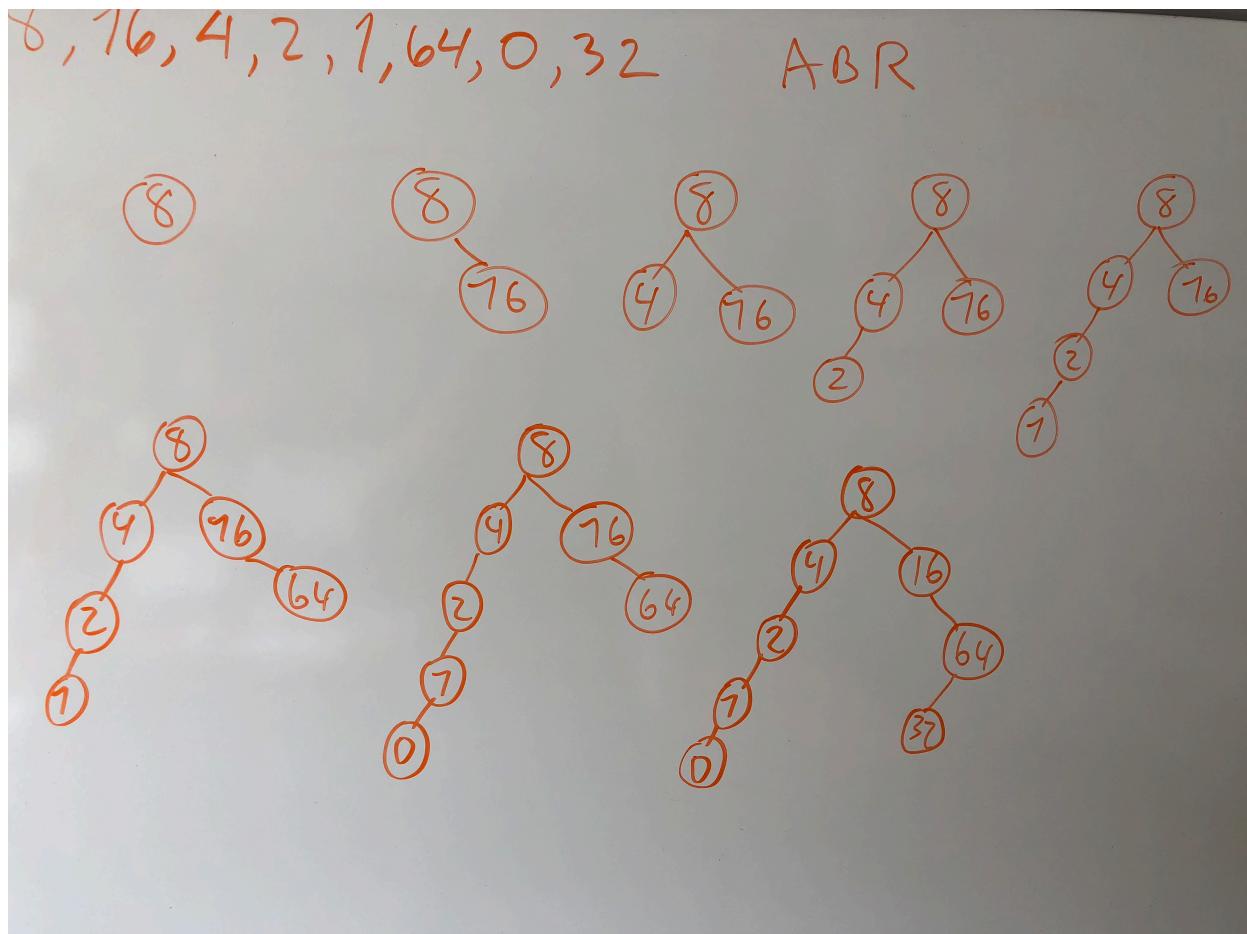
3. (50) Dessinez les arbres obtenus (8 au total) après l'insertion des clés { 8, 16, 4, 2, 1, 64, 0, 32 }, dans cet ordre, dans un arbre initialement vide de type :

a) (6) Monceau

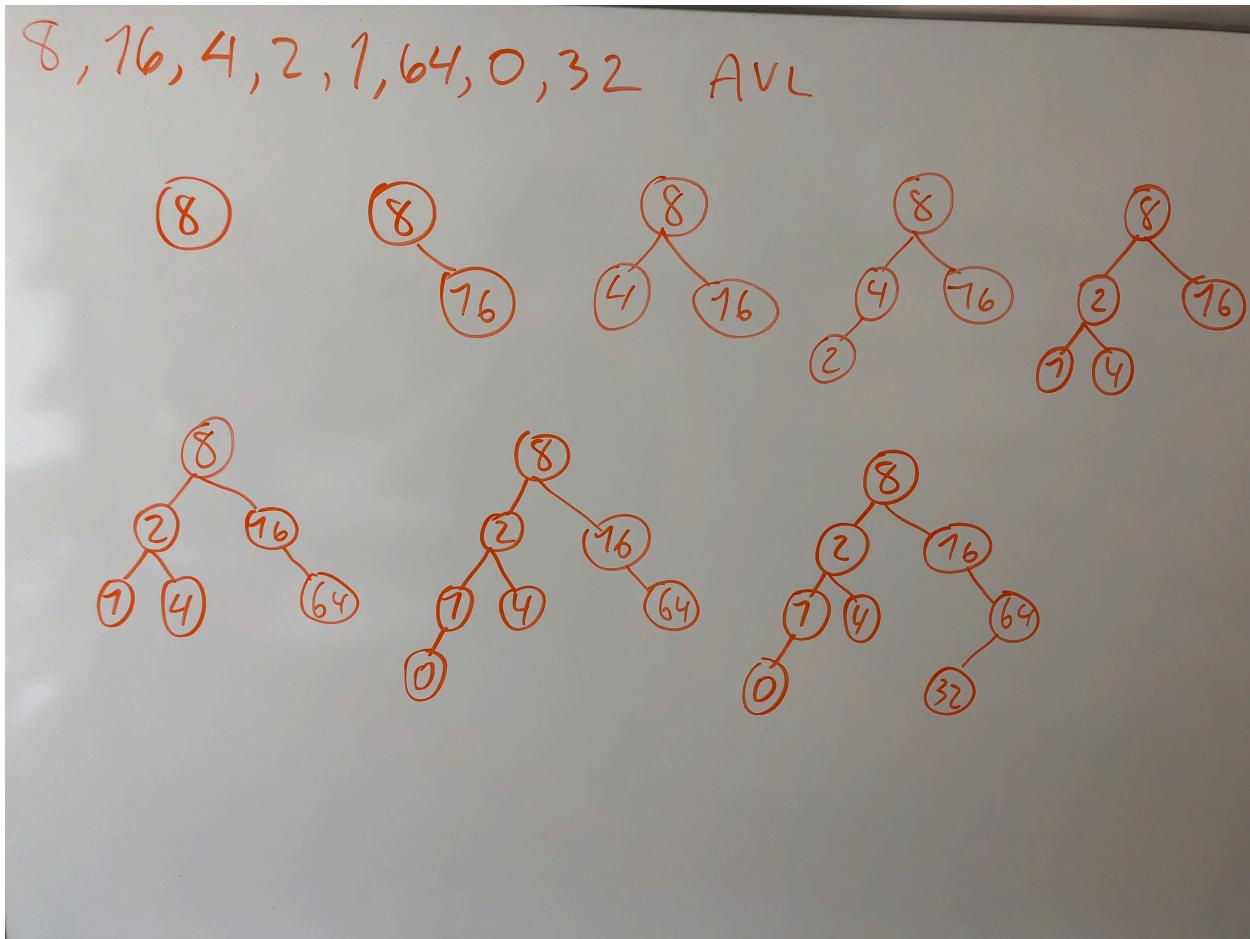


ON ACCEPTE UN HEAP-MAX, puisque non spécifié dans la question.

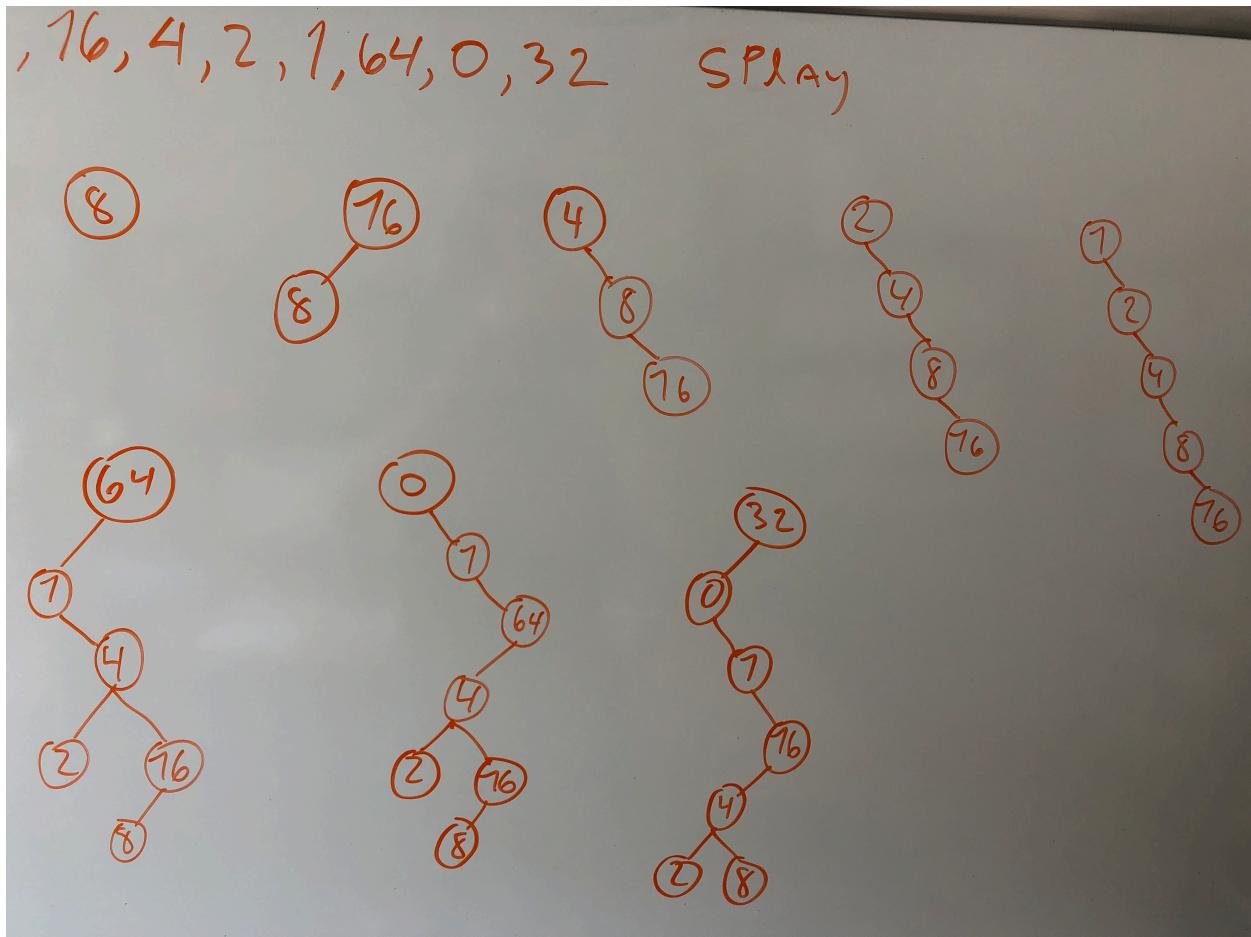
b) (6) Arbre binaire de recherche



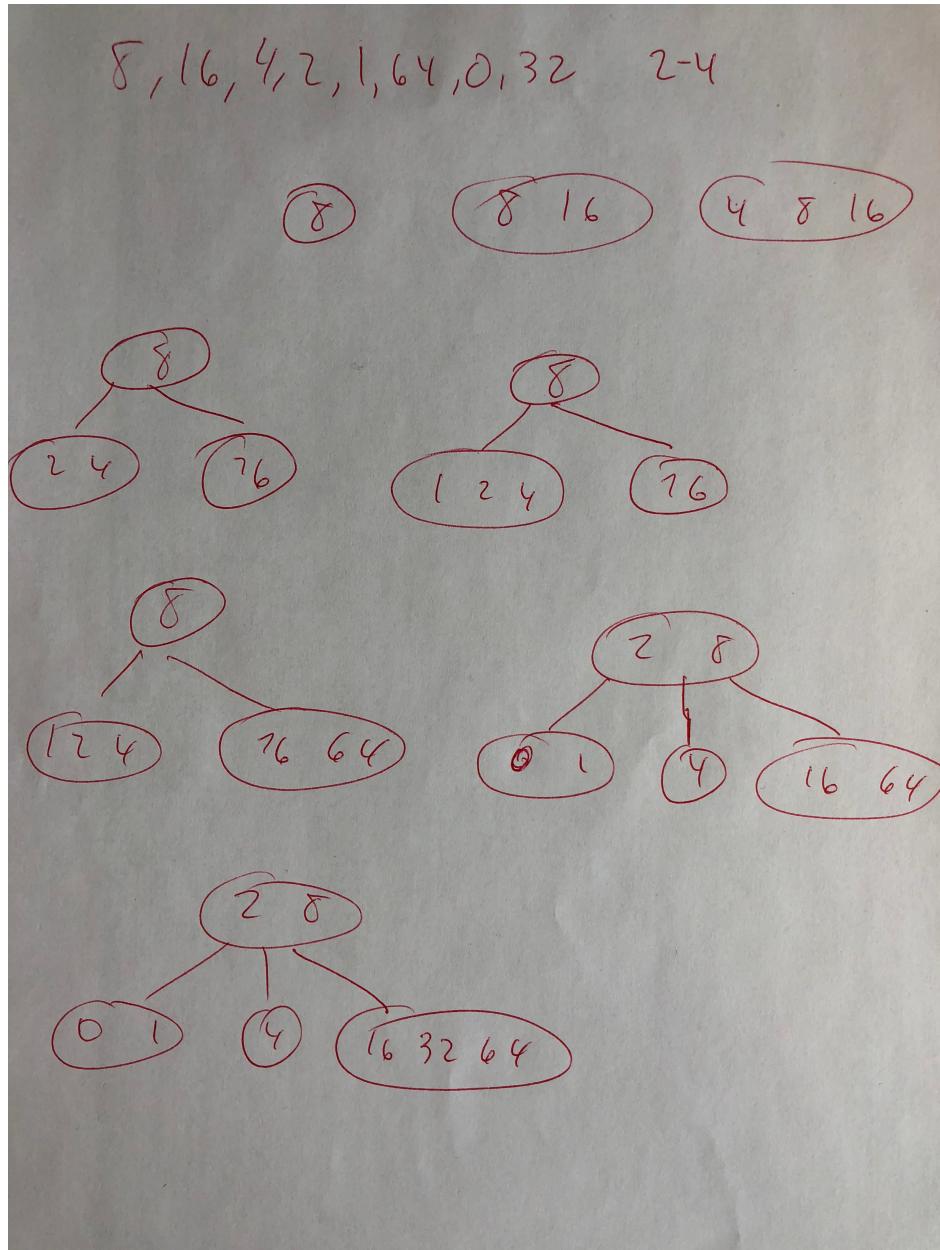
c) (6) Arbre AVL



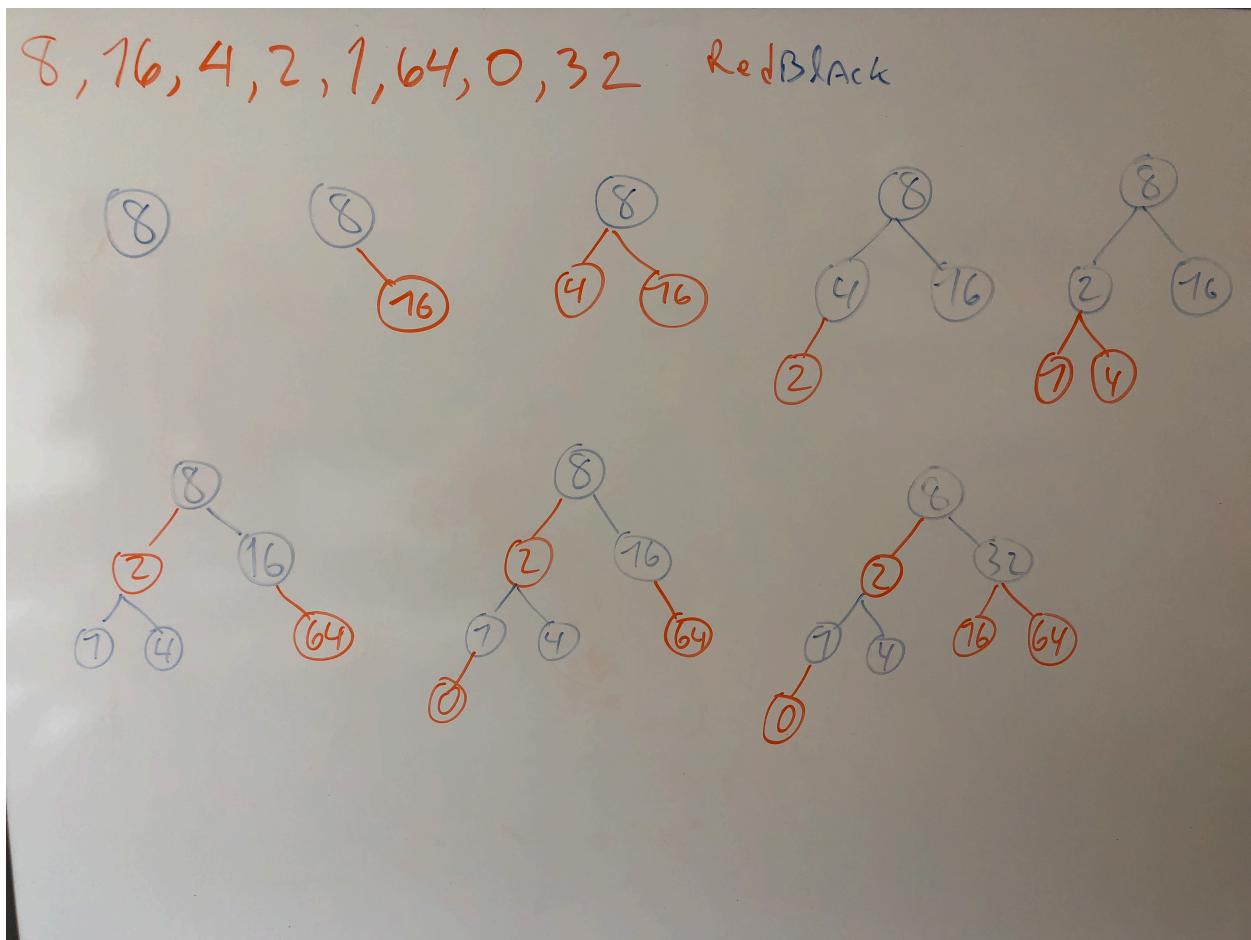
d) (6) Arbre "Splay"



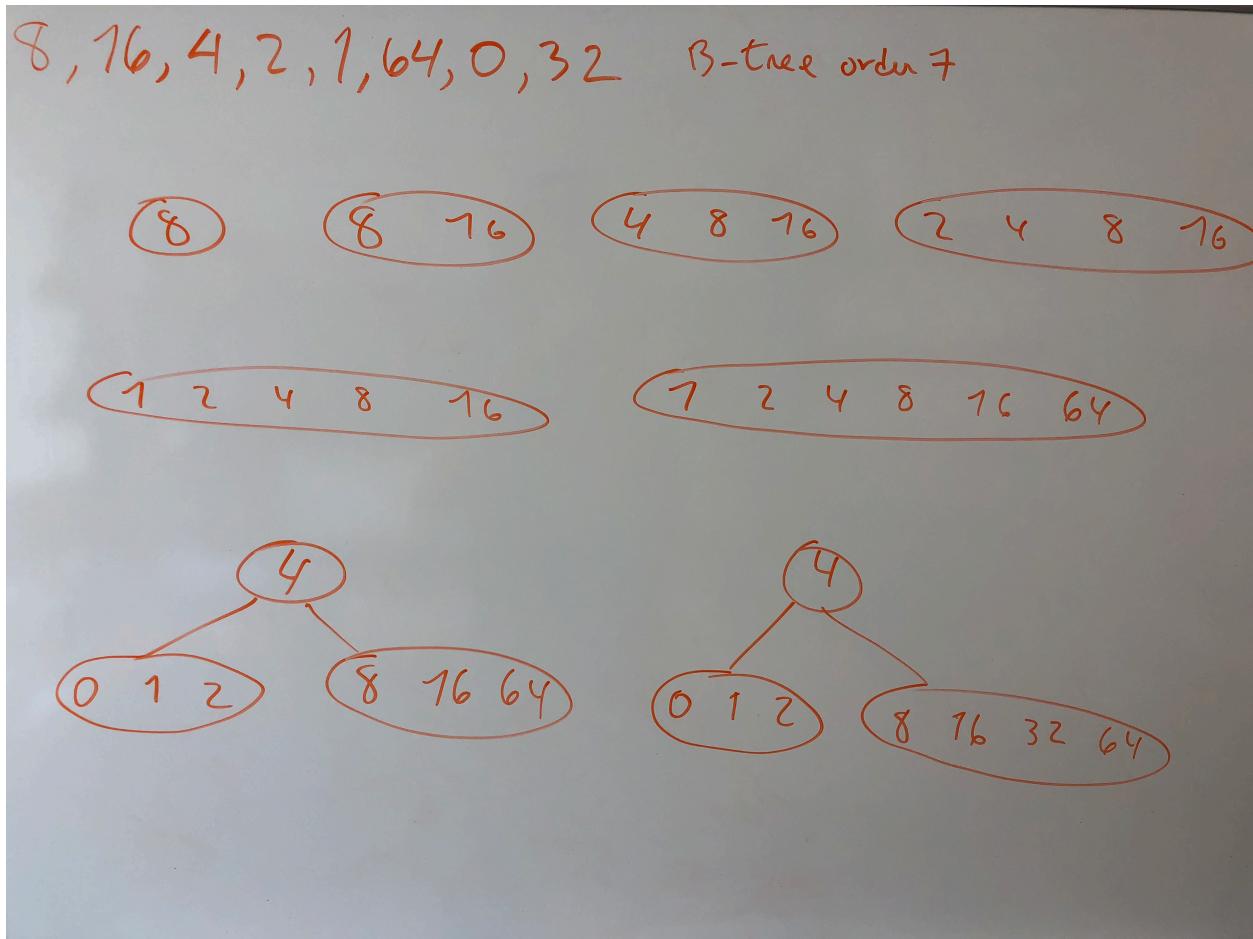
e) (8) Arbre 2-4



f) (8) Arbre rouge-noir



- g) (10) Arbre-B d'ordre 7 (un arbre-B d'ordre d est un arbre (a, b) où $a = d/2$ et $b = d$)



ON ACCEPTE AUSSI ($a = 3, b = 7$), puisque plafond/plancher n'était pas spécifié pour la valeur de a .

4. (40) Une **Map** bidirectionnelle est une **Map** qui prend en charge la recherche bidirectionnelle : à partir d'une clé, k , vous pouvez trouver la valeur, v , correspondante, et étant donnée une valeur, v , vous pouvez trouver la clé, k , correspondante. Dans une **Map** bidirectionnelle, il existe toujours une relation biunivoque entre les clés et les valeurs. En d'autres termes, chaque clé a exactement une valeur, et chaque valeur est trouvée sous exactement une clé. La **Map** bidirectionnelle prend en charge les opérations de **Map** habituelles (voir le tableau ci-dessous). Nous pouvons implémenter une **Map** bidirectionnelle en utilisant deux **Map**, chacune implémentée par un arbre rouge-noir, par exemple.

- *forward* est une **Map** des clés aux valeurs. Dans l'exemple ci-dessous, après les 3 premières opérations elle contient les entrées {1:2 et 3:4}.
- *back* est une **Map** des valeurs aux clés. Dans l'exemple ci-dessous, lorsque *forward* contient les entrées {1:2 et 3:4}, *back* contient les entrées {4:3 et 2:1}.

L'invariant est que les deux **Map** contiennent toujours les mêmes données: *forward* contient le mapping $k \rightarrow v$, si et seulement si *back* contient le mapping $v \rightarrow k$. On peut implémenter le “*lookup*”, $M[k]$, soit la recherche de la clé k dans *forward* et “*rlookup*”, la recherche renversée, comme la recherche de la clé v dans *back*.

Votre tâche consiste à implémenter les opérations restantes, **insérer** et **supprimer**, avec une complexité de temps d'exécution dans $O(\log n)$. ATTENTION : l'algorithme est plus compliqué qu'il n'y paraît (e.g. opération #5 dans le tableau ci-bas). Veillez à conserver l'invariant de la structure de données. C'est aussi une bonne idée de tester votre solution sur l'exemple. Donnez le pseudo-code pour chacune des opérations. Vous n'avez pas besoin d'écrire du code Python, mais soyez précis - un programmeur compétent devrait être capable de prendre votre description et de l'implémenter facilement.

Vous pouvez utiliser librement les structures de données standards et les algorithmes du cours dans votre solution, y compris pour l'insertion, la recherche et la suppression dans une **Map**, sans expliquer comment ils sont implémentés.

| | Opération | Résultat |
|---|---------------------------------|---|
| 1 | $m = \text{BidirectionalMap}()$ | {} |
| 2 | $m[1] = 2$ | {1:2} |
| 3 | $m[3] = 4$ | {1:2, 3:4} |
| 4 | $m[1]$ | 2 |
| 5 | $m[4] = 2$ | {3:4, 4:2} Notez que 1:2 est remplacé par 4:2. |
| 6 | $m.rlookup(2)$ | 4 |
| 7 | $\text{del } M[4]$ | {3:4} |

```
def __setitem__( self, k, v ):  
  
    vv = self.forward[k]  
    if vv:  
        del self.forward[k]  
        del self.back[vv]  
    kk = self.back[v]  
    if kk:  
        del self.forward[kk]  
        del self.back[v]  
    self.forward[k] = v  
    self.back[v] = k
```

```
def __delitem__( self, k ):  
  
    v = self.forward[k]  
    if v:  
        del self.forward[k]  
        del self.back[v]
```

5. (30) Un labyrinthe est ***construit correctement*** s'il existe un chemin depuis le départ jusqu'à la fin, tout le labyrinthe est accessible depuis le départ et il n'y a pas de boucle. On représente un labyrinthe par une matrice booléenne $n \times n$. Considérez les 2 exemples suivants, où le départ est en haut à gauche et la fin en bas à droite (en caractères gras et soulignés) et les 1 sont reliés par un chemin. Les 1 en caractères gras dans le labyrinthe correct est un chemin du départ à la fin.

```
correct = [[1,1,1,1,0],           incorrect = [[1,1,0,0,0],
    [0,1,0,1,0],                   [0,1,0,0,1],
    [1,0,0,1,1],                   [1,0,0,1,1],
    [0,1,0,0,0],                   [0,0,0,0,0],
    [1,0,1,1,1]]                  [1,0,1,0,1]]
```

- a) (20) Comment pouvons-nous déterminer si un labyrinthe est construit correctement ?

Pour n' sommets, $n \leq n' \leq n^2$, et m arêtes, $n \leq m \leq 8n$.

1. Transformer la matrice booléenne 2D en une structure de graphe dirigé, G ; $O(n^2)$
2. Appliquer DFS à partir du sommet de départ, ce qui donne l'arbre DFS enraciné au sommet de départ, T ; $O(n' + m)$, donc en pire cas dans $O(n^2)$.
3. Trouver un chemin entre les sommets de départ et fin avec la fonction ***construct_path*** qui prend en entrée l'arbre DFS T ; $O(n')$, donc en pire cas dans $O(n^2)$.
4. Déterminer la connectivité du graphe, càd si la taille de $T = n'$; $O(1)$

- b) (10) Quelle est la performance en temps d'exécution de votre algorithme ?

La pire étape est de construire le graphe, c'ad lire la matrice booléenne 2D, donc **$O(n^2)$** !!! Sinon, si le graphe est construit correctement, l'application de DFS qui est dans $O(n' + m)$, donc, encore une fois, dans $O(n^2)$.