Nom :_____

Code permanent : _____

Examen prédoctoral, cochez ici ☐

Directives pédagogiques :

- Inscrivez votre nom et code permanent au haut de cette page.
- Lisez attentivement toutes les questions et **répondez directement sur le questionnaire**.
- Seule l'utilisation d'un crayon est permise, **aucune documentation, calculatrice, téléphone cellulaire, ordinateur, ou autre objet permis**.
- Cet examen contient 8 questions pour 155 points au total (estimation de 1 point par minute environ).
- Cet examen contient 19 pages.
- Pour les questions à développement, **écrivez lisiblement et détaillez vos réponses**.
- Vous avez 160 minutes pour compléter cet examen.

BONNE CHANCE ET BON ÉTÉ !

1	/ 15
2	/ 20
3	/ 10
4	/ 15
5	/ 10
6	/ 30
7	/ 20
8	/ 35
Total	/155

1. (15) On veut développer un système pour gérer les allocations dynamiques de mémoire. On emballe les fonctions `malloc()` et `free()` avec ses propres fonctions qui enregistrent l'information d'une allocation dans un `bloc`. `malloc` est une fonction de la bibliothèque standard C permettant d'allouer dynamiquement de la mémoire. La libération de la mémoire ainsi réservée s'effectue avec la fonction `free`. Un `bloc` est constitué d'une adresse (celle retournée par `malloc`) et de la quantité de mémoire demandée en octets. Un `bloc` est ajouté à une collection à chaque allocation mémoire. Le `bloc` correspondant est retrouver et détruit à chaque dé-allocation mémoire. Proposez une structure de données pour gérer la collection de `bloc` qui garantirait que la performance d'un programme qui l'utilise se détériore le moins possible. Justifiez en quoi la structure de données que vous suggérez garantie sa performance.

Ça peut se faire avec une table de hachage en O(1) en moyenne pour l'insertion et la deletion. Par contre, O(n) en pire cas si les clés sont mal distribuées dans la table.

(accepter aussi)
Arbre binaire de recherche balancé, comme arbre rouge-noir, avec toutes les opérations garanties en O(log n), bien que moins bon qu'une table de hachage en moyenne.

2. Considérez les méthodes de tri que nous avons vu au cours.

a) (5) Qu'est-ce qu'une méthode de tri stable ?

Un algorithme de tri qui garde les éléments de clés égales dans le même ordre relatif en sortie qu'elles étaient en entrée.

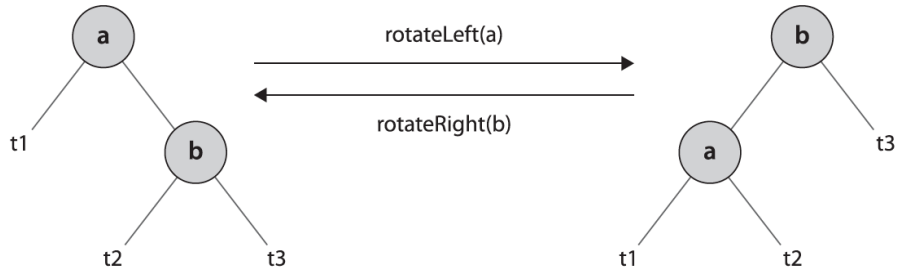
b) (15) Est-ce que le tri par monceau est stable et pourquoi ?

Le tri par monceau trouve l'élément le plus grand et le place à la fin de la liste. Donc, pour des clés égales, celles qui sont choisies en premier sont placées à la fin, et les autres occurrences seront placées devant.

3. (10) Complétez le code de la recherche séquentielle de l'élément `t` dans le tableau `A` de `n` éléments. La fonction retourne la valeur `vrai` si l'élément t se trouve dans le tableau ou `false` sinon.

```
recherche( A, t )

for( int i = 0 to n-1 )
    if( A[i] == t )
        return true
return false

end
```

4. Considérez les rotations gauche et droite d'arbres binaires de recherche :




```
public class RBTNode {

// Un RBTNode contient son element (element), deux references
// sur ses enfants gauche (gauche) et droite (droite) et une couleur.
// Pour tout x dans le sous-arbre gauche: x.compareTo(element) < 0
// Pour tout y dans le sous-arbre droit:  y.compareTo(element) > 0

    protected Comparable element;
    protected boolean couleur;
    protected RBTNode gauche, droite;

    private static final boolean Red = true;
    private static final boolean Black = false;
```

a) (5) Implantez "rotateRight" (PS. pseudo-code ok).

```
  private RBTNode rotateRight( RBTNode h ) {

      RBTNode x = h.gauche;
      h.gauche = x.droite;
      x.droite = h;
      return x;

  }
```

b) (5) Implantez "rotateLeft" (PS. pseudo-code ok).

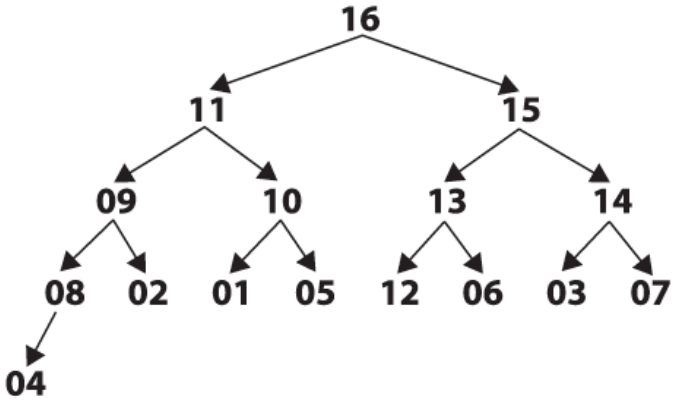
```
private RBTNode rotateLeft( RBTNode h ) {

    RBTNode x = h.droite;
    h.droite = x.gauche;
    x.gauche = h;
    return x;

}
```

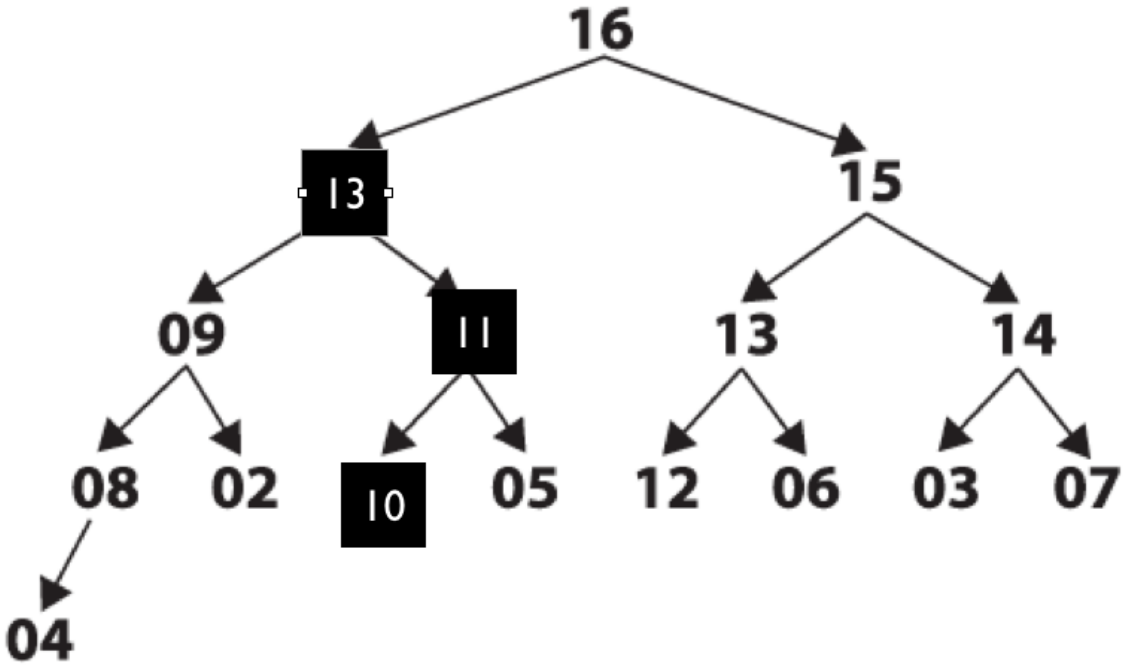
c) (5) Quelles sont les complexités des opérations que vous avez implantées en (a) et (b) ?

O(1) dans les 2 cas.

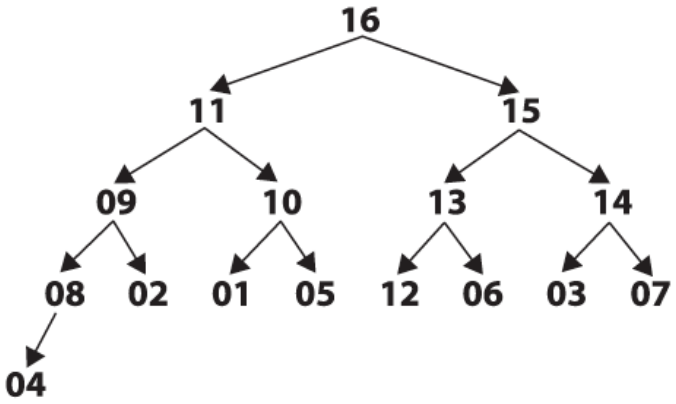
5. Considérez la structure en monceau.

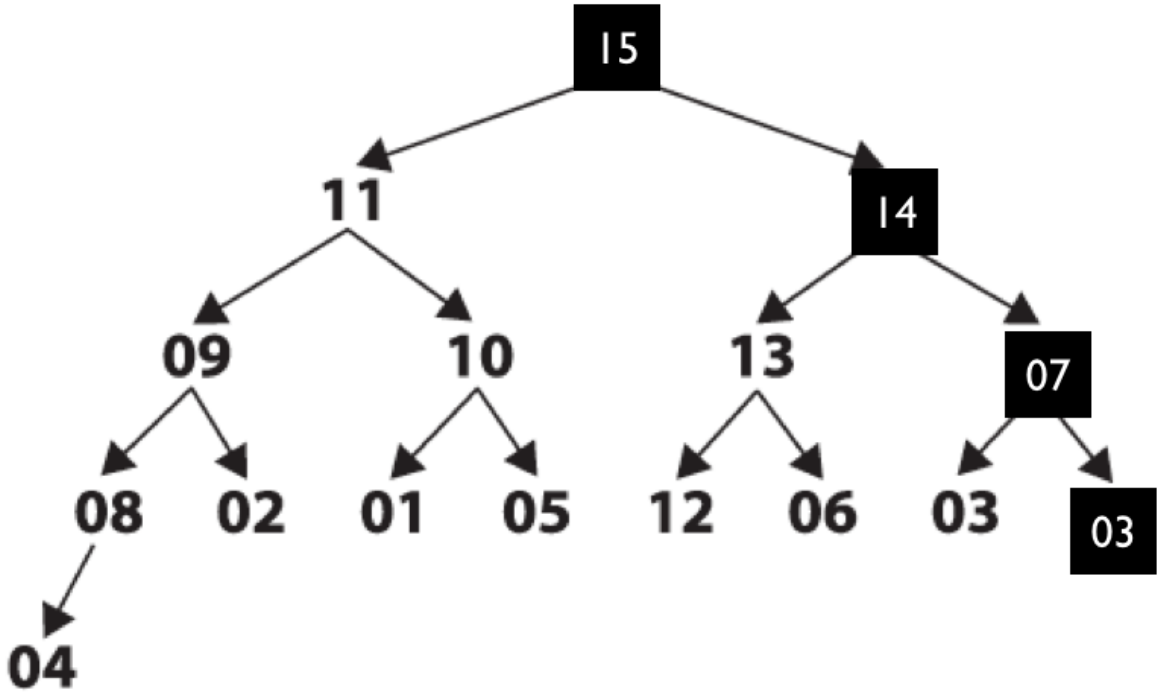
a) (5) Dessinez le monceau résultant du changement de la priorité 01 par 13 dans le monceau suivant.





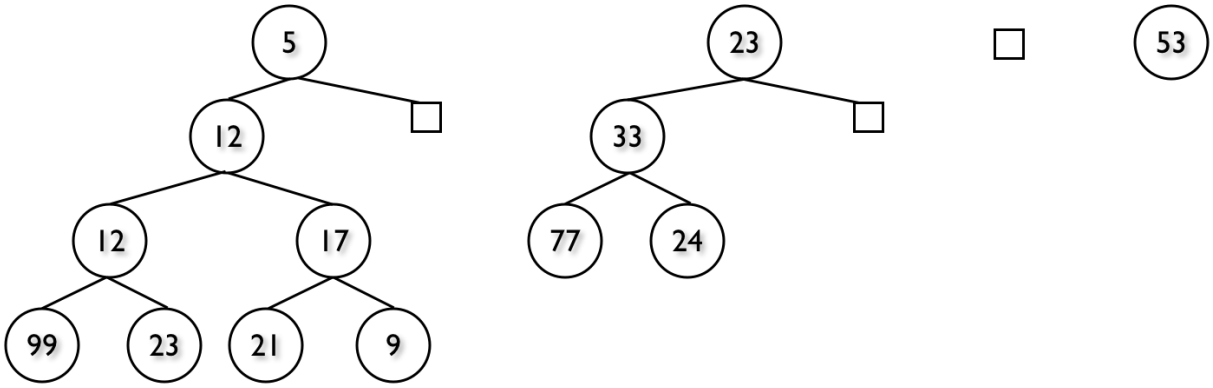
b) (5) Dessinez le monceau résultant du changement de la priorité 16 par 03 dans le monceau suivant.



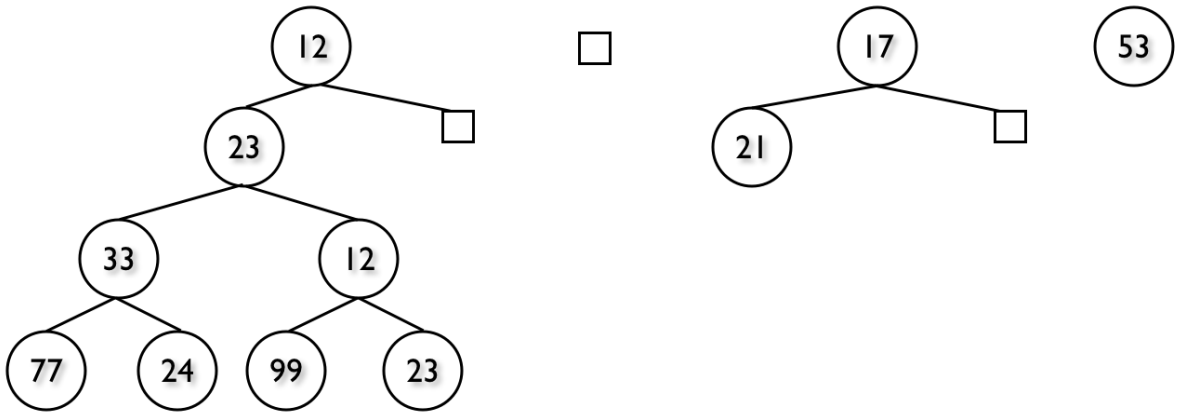


6. Considérez les queues binômiales pour gérer une collection avec priorités minimum, soit plus une clé est petite et plus elle est prioritaire.

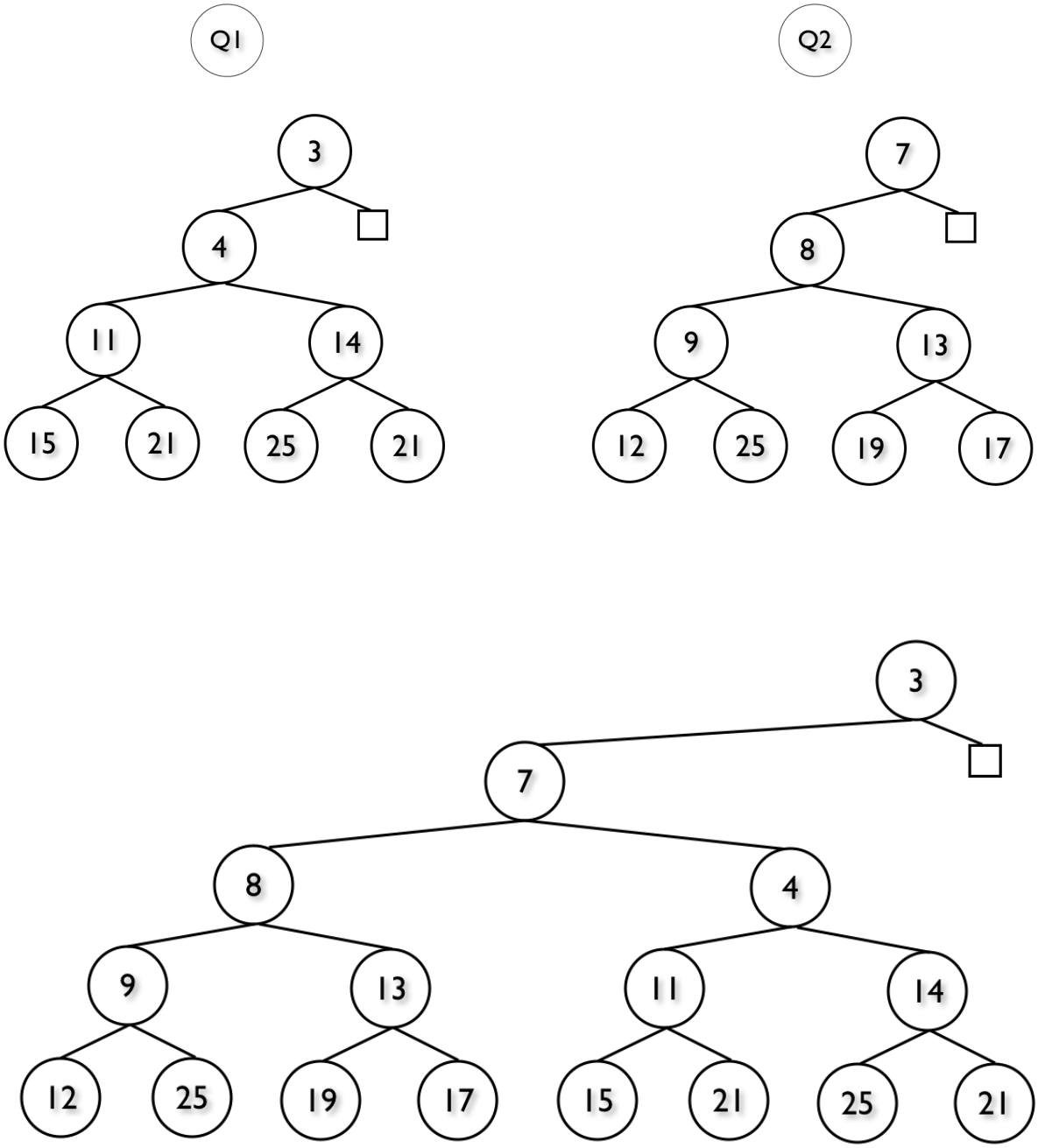
   a) (10) Dessinez la queue binômiale min résultante des insertions des clés suivantes dans cet ordre : 9, 5, 17, 21, 99, 12, 23, 12, 77, 33, 24, 23, 53.



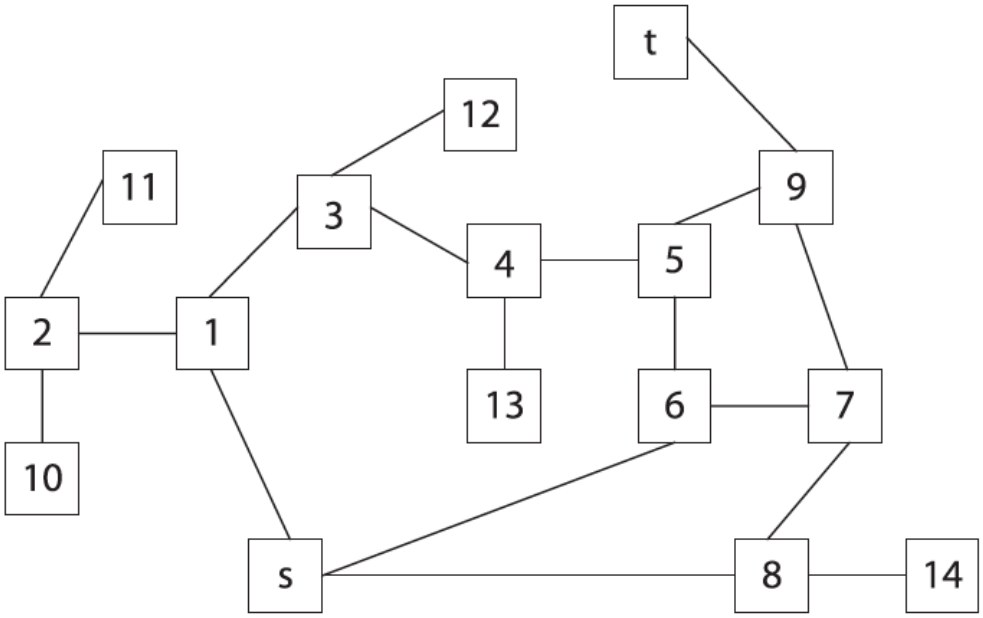
b) (10) Dessinez la queue binômiale min résultante de 2 opérations "getMin" consécutives sur la queue binômiale que vous avez obtenue en (a).



c) (10) Dessinez la queue binômiale min résultante de la fusion des queues, `Q1` et `Q2`, suivantes.



7. Considérez le graphe suivant (s = 0; t = 15) :




```
v[0](W): [< 1, 0 >, < 6, 0 >, < 8, 0 >]
v[1](W): [< 0, 0 >, < 2, 0 >, < 3, 0 >]
v[2](W): [< 1, 0 >, < 11, 0 >, < 10, 0 >]
v[3](W): [< 1, 0 >, < 12, 0 >, < 4, 0 >]
v[4](W): [< 3, 0 >, < 13, 0 >, < 5, 0 >]
v[5](W): [< 4, 0 >, < 6, 0 >, < 9, 0 >]
v[6](W): [< 0, 0 >, < 5, 0 >, < 7, 0 >]
v[7](W): [< 6, 0 >, < 8, 0 >, < 9, 0 >]
v[8](W): [< 0, 0 >, < 7, 0 >, < 14, 0 >]
v[9](W): [< 5, 0 >, < 7, 0 >, < 15, 0 >]
v[10](W): [< 2, 0 >]
v[11](W): [< 2, 0 >]
v[12](W): [< 3, 0 >]
v[13](W): [< 4, 0 >]
v[14](W): [< 8, 0 >]
v[15](W): [< 9, 0 >]
```

a) (10) Dans quel ordre les sommets de ce graphe seront visités si on applique une recherche en profondeur qui débute avec le sommet s ? La recherche en profondeur utilise une pile pour stocker les sommets à visiter. Montrez l'état de cette pile à chaque étape de la recherche.

```
[1]
[2, 1]
[11, 2, 1]
[2, 1]
[10, 2, 1]
[2, 1]
[1]
[3, 1]
[12, 3, 1]
[3, 1]
[4, 3, 1]
[13, 4, 3, 1]
[4, 3, 1]
[5, 4, 3, 1]
[6, 5, 4, 3, 1]
[7, 6, 5, 4, 3, 1]
[8, 7, 6, 5, 4, 3, 1]
[14, 8, 7, 6, 5, 4, 3, 1]
[8, 7, 6, 5, 4, 3, 1]
[7, 6, 5, 4, 3, 1]
[9, 7, 6, 5, 4, 3, 1]
[15, 9, 7, 6, 5, 4, 3, 1]
[9, 7, 6, 5, 4, 3, 1]
[7, 6, 5, 4, 3, 1]
[6, 5, 4, 3, 1]
[5, 4, 3, 1]
[4, 3, 1]
[3, 1]
[1]
[]
```

b) (10) Même question qu'en (b) si on applique une recherche en largeur ? La recherche en largeur utilise une file pour stocker les sommets à visiter. Montrez l'état de cette file à chaque étape de la recherche.

```
Front [0] Back
Front [0, 1] Back
Front [0, 1, 6] Back
Front [0, 1, 6, 8] Back
Front [1, 6, 8] Back
Front [1, 6, 8, 2] Back
Front [1, 6, 8, 2, 3] Back
Front [6, 8, 2, 3] Back
Front [6, 8, 2, 3, 5] Back
Front [6, 8, 2, 3, 5, 7] Back
Front [8, 2, 3, 5, 7] Back
Front [8, 2, 3, 5, 7, 14] Back
Front [2, 3, 5, 7, 14] Back
Front [2, 3, 5, 7, 14, 11] Back
Front [2, 3, 5, 7, 14, 11, 10] Back
Front [3, 5, 7, 14, 11, 10] Back
Front [3, 5, 7, 14, 11, 10, 12] Back
Front [3, 5, 7, 14, 11, 10, 12, 4] Back
Front [5, 7, 14, 11, 10, 12, 4] Back
Front [5, 7, 14, 11, 10, 12, 4, 9] Back
Front [7, 14, 11, 10, 12, 4, 9] Back
Front [14, 11, 10, 12, 4, 9] Back
Front [11, 10, 12, 4, 9] Back
Front [10, 12, 4, 9] Back
Front [12, 4, 9] Back
Front [4, 9] Back
Front [4, 9, 13] Back
Front [9, 13] Back
Front [9, 13, 15] Back
Front [13, 15] Back
Front [15] Back
Front [] Back
```

8. Considérez un arbre de recherche pour un jeu où à chaque coup un joueur peut jouer en moyenne `b` coups différents. Considérez les méthodes de recherche dans les arborescences de jeux classiques, soit la recherche en profondeur, en largeur et A*.

a) (5) Combien y a-t-il d'états de ce jeu dans un arbre de profondeur `p` ?

$$\sum_{i=0}^{p} b^i$$

b) (5) Combien d'états différents au minimum devront être visités par une recherche en profondeur si un état gagnant du jeu se trouve à profondeur `p`.

p

c) (5) Même question qu'en (b) mais pour une recherche en largeur.

$$\sum_{i=0}^{p-1} b^i + 1$$

d) (5) Considérez la recherche A*. En vos mots, qu'est-ce qu'une heuristique admissible ?

Permet de guider la recherche A* vers une solution rapidement.

(optionnel)
Si en plus d'être admissible l'heuristique peut garantir que le coût effectué jusqu'à l'état courant est >= au coût réel, alors cette heuristique va aussi garantir une solution optimale en nombre de coups à jouer.

e) (5) Dans le cas où on utilise une heuristique admissible avec A*, combien d'états différents du jeu au minimum devront être visités si un état gagnant se trouve à profondeur `p` ?

`p`

f) (5) Même question qu'en (e) mais pour une recherche A* qui n'utilise pas une heuristique admissible.

`p`

g) (5) Même question qu'en (f) mais dans le pire cas, c'est-à-dire le nombre d'états différents du jeu qui pourrait être visités au maximum.

$$\sum_{i=0}^{p-1} b^i + 1$$