Nom :_____

Code permanent : _____

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 9 questions pour 120 points au total.
- Cet examen contient 19 pages.
- Pour les questions à développement, **écrivez lisiblement et détaillez vos réponses**.
- Vous avez 100 minutes pour compléter cet examen.

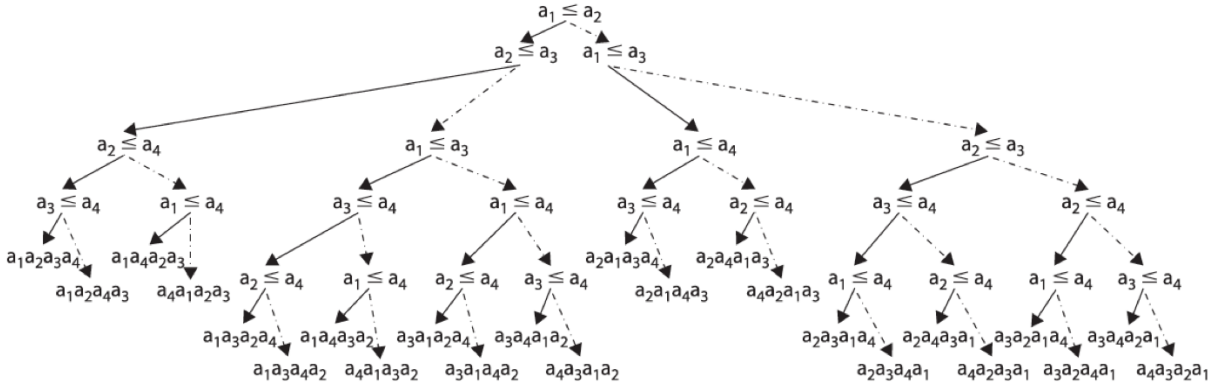
BONNE CHANCE !

1	/ 10
2	/ 10
3	/ 10
4	/ 25
5	/ 15
6	/ 12
7	/ 14
8	/ 12
9	/ 12
Total	/120

1. (10) Cochez la case des énoncés qui sont vrais.

a)	Ajouter de l'information à l'encodage des données d'un problème sauve du temps de calcul.	
b)	Changer d'ordinateur ne peut changer le temps d'exécution d'un programme que par un facteur constant.	X
c)	Des mesures empiriques de temps d'exécution révèle toujours des surprises.	
d)	O(n!) est dans $O(2^n)$ mais $O(2^n)$ n'est pas dans O(n!).	
e)	On peut toujours déterminer le comportement d'un algorithme en regardant le code du programme qui l'implante.	
f)	Les temps d'exécution d'une implantation récursive d'un algorithme seront toujours plus courts que ceux d'une implantation non-récursive.	
g)	Insérer un nouvel élément dans un tableau de n éléments se fait en moyenne en O(n/2) = O(n). (si spécifié trié, sinon c'est faux)	X
h)	La suppression d'un élément dans un tableau de n éléments se fait en moyenne en $O(n^2/4) = O(n)$.	
i)	La suppression du dernier élément d'une liste simplement chaînée implantée avec des pointeurs sur les premier et dernier éléments se fait en O(1).	
j)	Avec un pire cas dans O(log n) pour une collection de n éléments, la recherche dichotomique est un bien meilleur choix qu'un arbre binaire de recherche lorsqu'on devra faire beaucoup plus de recherche d'éléments que toute autre opération.	

2. (10) Démontrez qu'il est impossible de trier une collection de données à l'aide de comparaisons de 2 éléments de cette collection en moins que O(n log n) comparaisons.



La hauteur d'un arbre binaire de décisions pour un tri de 4 éléments est 5, correspondant au nombre nécessaire de comparaisons à effectuer pour arriver à la bonne permutation.

Comme on a n! permutations possibles pour n éléments, alors on aura au moins h comparaisons, soit la hauteur d'un arbre binaire de n! feuilles.

Pour un arbre binaire de décisions de n éléments, on a $h = 2^h-1$ noeuds et une hauteur de $h = \log (n+1)$;

si l'arbre n'est pas balancé, on sait que $h \geq \lceil \log(n+1) \rceil$, soit

$h = \lceil \log(n+1) \rceil$, dans le cas balancé, ou $h > \lceil \log(n+1) \rceil$ sinon.

Tout arbre binaire avec n! feuilles démontre directement qu'il possède au moins n! noeuds, simplement en calculant $h= \lceil \log(n!) \rceil$

Prenons les propriétés suivantes des logarithmes :

$\log(a*b) = \log(a)+\log(b)$; et, $\log(x^y) = y * \log(x)$

$h = \lceil \log(n!) \rceil \geq \log(n*(n-1)*(n-2)*... *2*1)$

$h \geq \log((n/2)^{n/2})$

$h \geq n/2 \log(n/2)$

$h \in O(n \log n)$

3. (10) Dessinez l'arbre des appels récursifs associé au tri rapide de la collection suivante si on choisit toujours l'élément du milieu du tableau comme pivot, soit celui à l'index ⌊(max+min+1)/2⌋, où min et max sont les bornes de la région du tableau à trier.

15	9	8	1	4	11	7	12	3	index pivot = ⌊9/2⌋ = 4
15	9	8	1	3	11	7	12	4	swap pivot
1	3	4	15	9	11	7	12	8	partition, pi = 2
1	3	4	15	9	11	7	12	8	pivots = 1, 6
1	4	3	15	9	11	8	12	7	swap pivots
1	3	4	7	9	11	8	12	15	partition, pi = 1, 3
1	3	4	7	9	11	8	12	15	pivot = 6
1	3	4	7	9	11	15	12	8	swap pivot
1	3	4	7	8	11	15	12	9	partition, pi = 4
1	3	4	7	8	11	15	12	9	pivot = 7
1	3	4	7	8	11	15	9	12	swap pivot
1	3	4	7	8	11	9	12	15	partition, pi = 7
1	3	4	7	8	11	9	12	15	pivot = 6, 8
1	3	4	7	8	11	9	12	15	swap pivot
1	3	4	7	8	9	11	12	15	partition, pi = 5, 8
1	3	4	7	8	9	11	12	15	

4. Considérez la collection suivante :

15	9	8	1	4	11	7	12	3

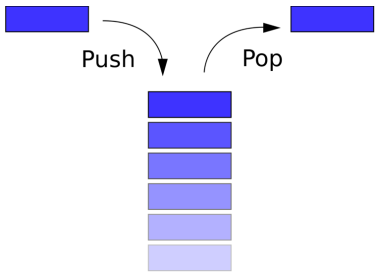
a) (10) Dessinez le monceau résultant de l'application du programme "buildHeap" suivant :

```
buildHeap (A)
1.    for i = ⌊n/2⌋ − 1 downto 0 do
2.      heapify (A, i, n)
end

heapify (A, idx, max)
1.    left = 2*idx + 1
2.    right = 2*idx + 2
3.    if (left < max and A[left] > A[idx]) then
4.      largest = left
5.    else largest = idx
6.    if (right < max and A[right] > A[largest]) then
7.      largest = right
8.    if (largest ≠ idx) then
9.      swap A[i] and A[largest]
10.     heapify (A, largest, max)
end
```

b) (5) Dessinez le tableau correspondant au monceau

c) (10) Expliquez en vos mots le pire cas du tri rapide et pourquoi le tri par monceau ne peut pas tomber dans ce cas et reste en O(n log n) en pire cas.

5. Soit une pile :



a) (5) Proposez une structure de données basée sur un tableau qui permettrait les opérations "push" et "pop" en O(1) lorsqu'il y a assez de place dans le tableau pour stocker tous les éléments. Décrivez votre structure de données en complétant le code de la **class** ArrayStack (PS. pseudo-code ok). Voir l'Appendice pour l'interface du type abstrait **Stack**.

```
public class ArrayStack implements Stack {
    // Implantation d'une pile avec un tableau.
    // top pointe l'element sur le dessus de la pile.

  private Object[] elems;
  private int top;
  private int espace;

  /////////////// Constructeur ////////////

  public ArrayStack( int espaceInitial ) {
    // Construit une pile, initialement vide.
    // Extension dynamique de la longeur du tableau,
    // fixee initialement a un espace initial.
    elems = new Object[espaceInitial];
    espace = espaceInitial;
    top = -1;
  }

}
```

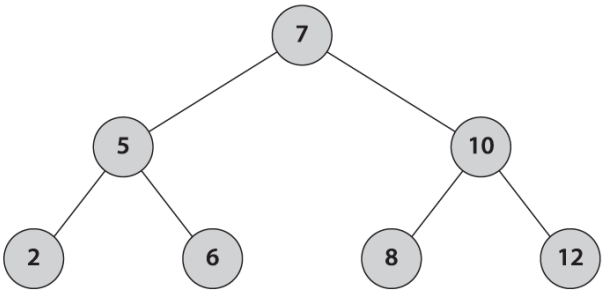
b) (5) Implantez les opérations "push", "pop" et "top" de votre pile (PS. pseudo-code ok).

```
  public void push( Object x ) {
    //ajoute un objet sur le top de la pile.

if( top == espaceInitial - 1 ) {
  nouvelEspace = 2 * espace;
  Object[] nouvelElems = new Object[nouvelEspace];
  System.arraycopy( elems, 0, nouvelElems, 0, espace );
  elems = nouvelElems;
  espace = nouvelEspace;
}
elems[++top] = x;




  }


   public Object pop() {
     //retourne et enleve le top de la pile.

     if( top == -1 ) return null;
     else return( elems[top--] );







   }

   public Object top() {
     //retourne l'objet sur le top de la pile.

     if( top == -1 ) return null;
     else return( elems[top] );




   }
```

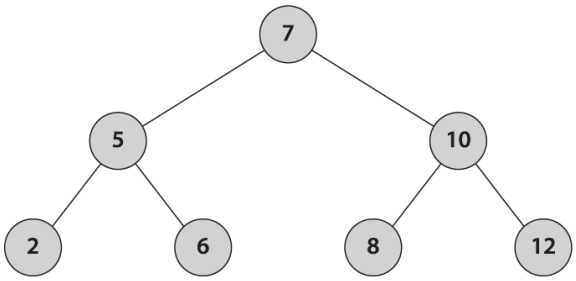
c) (5) Quelle serait la complexité de l'opération "push" lorsqu'il n'y a plus de place pour insérer un nouvel élément sur la pile ?

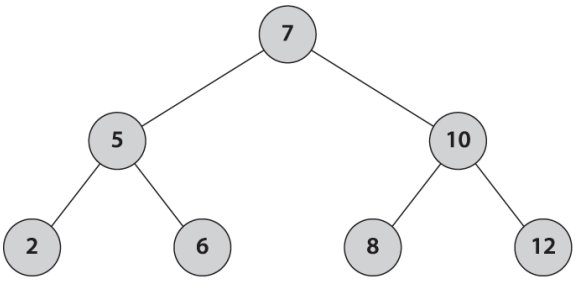
O(n); pour copier la pile dans un nouveau tableau.

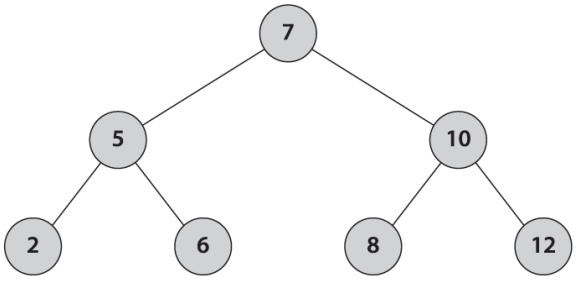
6. Considérez l'arbre binaire de recherche suivant :

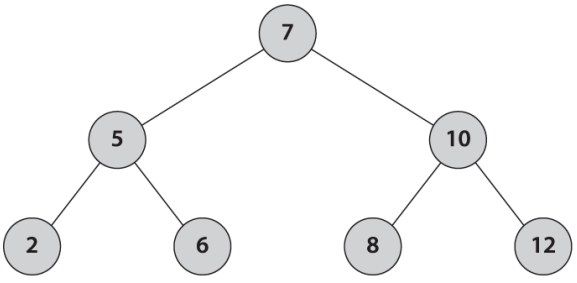


a) (8) Dessinez **les** arbres obtenus après chaque insertion des valeurs 9, 4, 11 et 13 (dans cet ordre).

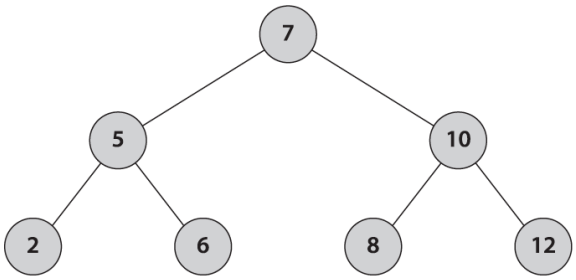


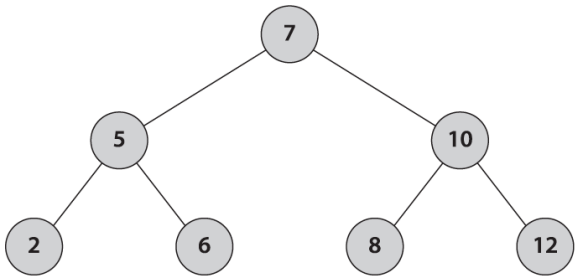


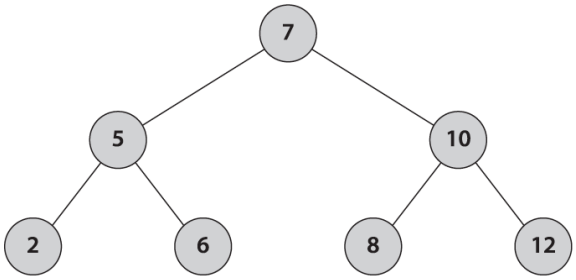


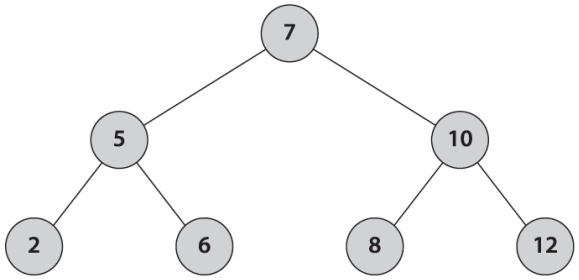


b) (4) De votre dernier arbre obtenu en (a), dessinez chaque arbre obtenu après la suppression des valeurs 10 et 7 (dans cet ordre).

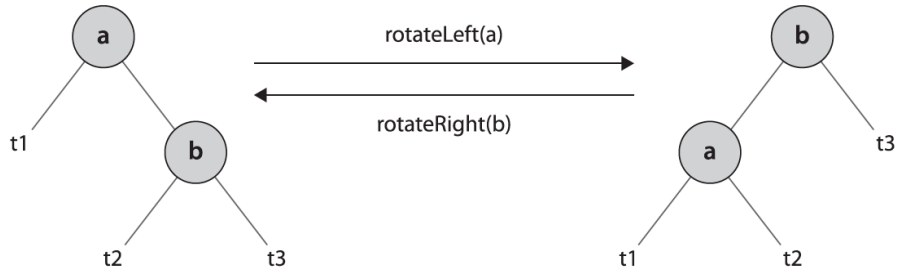








7. Considérez les rotations gauche et droite d'arbres binaires de recherche :




```
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) (4) Quelle est la complexité des opérations que vous avez implantées en (a) et (b) ?

O(1)

8. Considérez une table de hachage implantée dans un tableau de N listes simplement chaînées (N > 1) et la fonction hash(e) = e mod N.

a) (8) Dessinez la table résultante des insertions e = 1, 4, 8, 9, 11, 15 et 17 (dans cet ordre) lorsque N = 4.

b) (2) Quelles sont les complexités en meilleur cas, moyenne et pire cas de l'insertion ?

c) (2) Quelles sont les complexités en meilleur cas, moyenne et pire cas de la suppression ?

9. Considérez le stockage d'une très grande collection de fichiers et une fonction permettant la recherche d'un fichier par son nom. On connait la taille de la collection, N. Vous devrez faire de nombreuses recherches et peu de mises à jour (insertions et suppressions). Le nom du fichier sert de clé de recherche. Décrivez en vos mots les pour et les contres d'utiliser les structures de données suivantes pour gérer cette collection :

   a) (2) Un tableau de taille N trié avec recherche dichotomique.

   b) (2) Une pile de taille N.

c) (2) Un arbre binaire de recherche.

d) (2) Un arbre binaire rouge-noir.

e) (2) Une table de hachage de N listes simplement chaînées.

f) (2) Quelle structure de données proposeriez vous pour gérer cette collection si dans une autre application les nombres de recherche et de mises à jour étaient à peu près égaux (#recherches ~= #mises à jour) et pourquoi ?

Appendice : type abstrait **Stack**

```
import java.util.Iterator;

public interface Stack {

    // Methodes d'acces ...

    public boolean isEmpty();
    // Retourne true ssi la pile est vide.

    public int size();
    // Retourne la longueur de la pile.

    // Methodes de transformation ...

    public void clear();
    // Vide la pile.

    public void push( Object x );
    // Empile x.

    public Object pop();
    // Depile,
    // retourne null si la pile est vide.

    // Iterateur ...
    public Iterator iterator();
    // Retourne un iterateur qui parcours les elements
    // de la pile, dans l'ordre du premier au dernier.
}
```