ÉCOLE POLYTECHNIQUE MONTRÉAL

Le génie sans frontières

Questionnaire examen intra

INF2010

Sigle du cours

Q1	
Q2	
Q3	
Q4	
Q5	
Q6	
Total	

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Sigle et titre du cours		*Groupe*	*Trimestre*
INF2010 – Structures de données et algorithmes		**Tous**	**20131**
Professeur		***Local***	***Téléphone***
Ettore Merlo, responsable – Tarek Ould Bachir, chargé de cours		**M-2004**	**7128**
Jour	***Date***	***Durée***	***Heure***
Jeudi	**21 février 2013**	**2h00**	**10h00**

Documentation	*Calculatrice*	
☐ Toute	☐ Aucune	**Les cellulaires, agendas électroniques et téléavertisseurs sont interdits.**
☒ Aucune	☐ Programmable	
☐ Voir directives particulières	☒ Non programmable	

Directives particulières

Bonne chance à tous!

Important

Cet examen contient **5** questions sur un total de **22** pages **(excluant cette page)**

La pondération de cet examen est de **30** **%**

Vous devez répondre sur : ☒ le questionnaire ☐ le cahier ☐ les deux

Vous devez remettre le questionnaire : ☒ oui ☐ non

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduite.

Question 1 : Tables de dispersement (15 points)

Soit une table de dispersement où les collisions sont gérées par débordement progressif avec sondage et où Hash(clé) = (clé + i^2 + 2 i + 1) % N :

1.1) **(2 point)** Quelle est la complexité asymptotique en insertion de cette structure de données (considérez la complexité en cas moyen) ? Justifiez votre réponse.

1.2) **(2 point)** Quelle est la complexité asymptotique en insertion de cette structure de données si tous les éléments insérés généraient la même valeur Hash(clé) à la première insertion (i=0) ? Justifiez votre réponse.

1.3) **(2 points)** Compléter la fonction `findPos()` donnée à l'Annexe 1 et reproduite ci-après. Pour mémoire, la fonction doit implémenter Hash(clé) = (clé + i^2 + 2 i + 1) % N.

```
/**
  * Trouver la position de x
  */
private int findPos( AnyType x )
{
   int offset =  _________________; // VOTRE REPONSE ICI
   int currentPos = myhash( x );

   while( array[ currentPos ] != null &&
      !array[ currentPos ].element.equals( x ) )
   {
      currentPos += offset;  // Compute ith probe
      offset +=  _________________; // VOTRE REPONSE ICI
      if( currentPos >= array.length )
         currentPos -= array.length;
   }

   return currentPos;
}
```

1.4) **(5 points)** En vous servant du tableau ci-dessous, donnez l'état de la mémoire d'une table de taille N=13 après l'insertion des clés suivantes:

91, 56, 78, 65, 48.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Entrées													

1.5) **(4 points)** Si l'on continue à insérer de nouvelles clés, combien d'entrées la table contiendra-t-elle au moment où une nouvelle insertion provoquera un rehash ? Quelle sera la nouvelle taille de la table ? **Référez-vous au code Java donné à l'Annexe 1.**

Question 2 : Tris en $n \log(n)$ (24 points)

Partie 1:

On désire exécuter l'algorithme « MergeSort » donné à l'Annexe 2 pour trier le vecteur suivant.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

2.1) **(4 points)** Illustrez les étapes de l'exécution de l'algorithme en vous servant du tableau ci-dessous :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1
Blocs de 2																
Blocs de 4																
Blocs de 8																
Fin																

2.2) **(4 points)** Au total, quel est le nombre de fois que la fonction récursive `mergesort` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void mergesort(AnyType [ ] a, AnyType [ ] tempArray, int left, int right)
```

Votre réponse: _____

2.3) **(4 points)** Au total, quel est le nombre de fois que la fonction `merge` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void merge (AnyType [ ] a, AnyType [ ] tempArray, int leftPos,
            int rightPos, int rightEnd)
```

Votre réponse: _____

Partie 2:

On désire maintenant étudier l'exécution de l'algorithme « QuickSort » donné à l'Annexe 2 pour trier le même vecteur, reproduit ci-après. On considère une valeur *cut-off* de 3.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

2.4) (**2 points**) Donnez :

Les trois valeurs de « Median3 » à la première récursion :

La valeur de la médiane (pivot) :

2.5) (**2 points**) Donnez l'état du vecteur après l'exécution de Median3 de la première récursion :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs																

2.6) (**4 points**) Donnez l'état du vecteur après l'exécution du partitionnement de la première récursion :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs																

2.7) **(4 points)** Au total, quel est le nombre de fois que la fonction récursive `quicksort` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a, int left, int right )
```

Votre réponse: _____

Question 3 : Arbres binaire de recherche et arbres AVL (26 points)

Considérez les affichages des arbres binaires de recherche suivants.

a) Pour chaque arbre, donnez la représentation graphique de l'arbre.

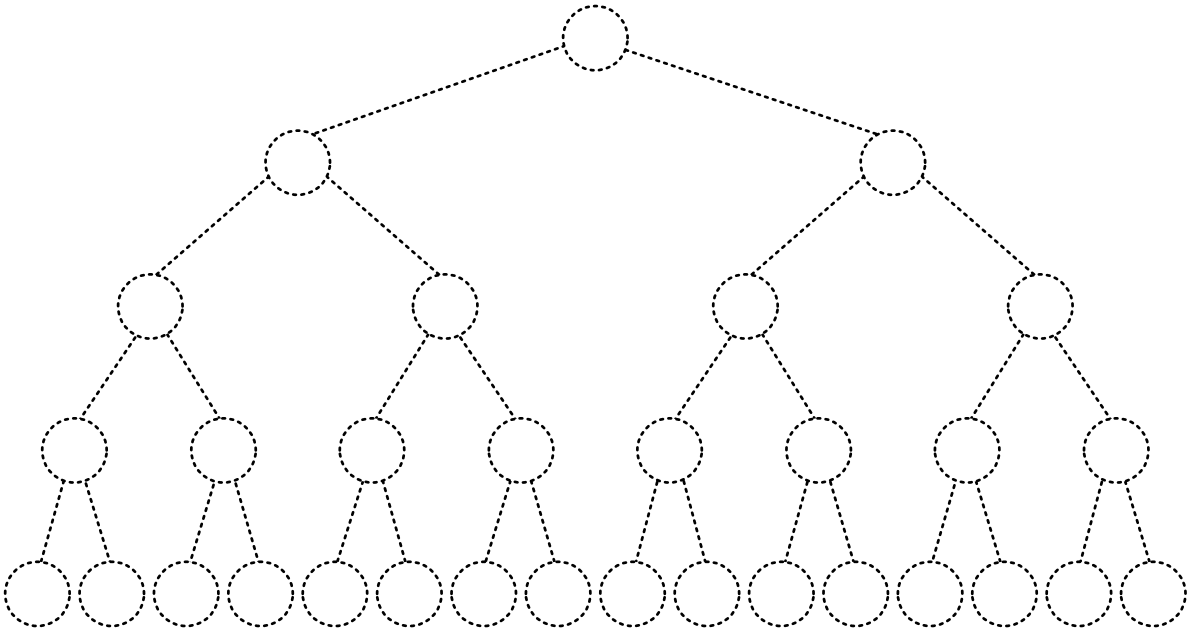
b) Insérez un nœud contenant la clé **31** dans l'arbre binaire de recherche.

c) Si l'arbre obtenu en (a) n'est pas un AVL, indiquez-le et passez à la question suivante. Si au contraire l'arbre binaire obtenu en (a) est un AVL, appliquez les rotations appropriées à l'arbre obtenu en (b) pour qu'il redevienne un AVL (autrement dit, exécutez une insertion de type AVL de la clé **31** dans l'arbre obtenu en (a)).

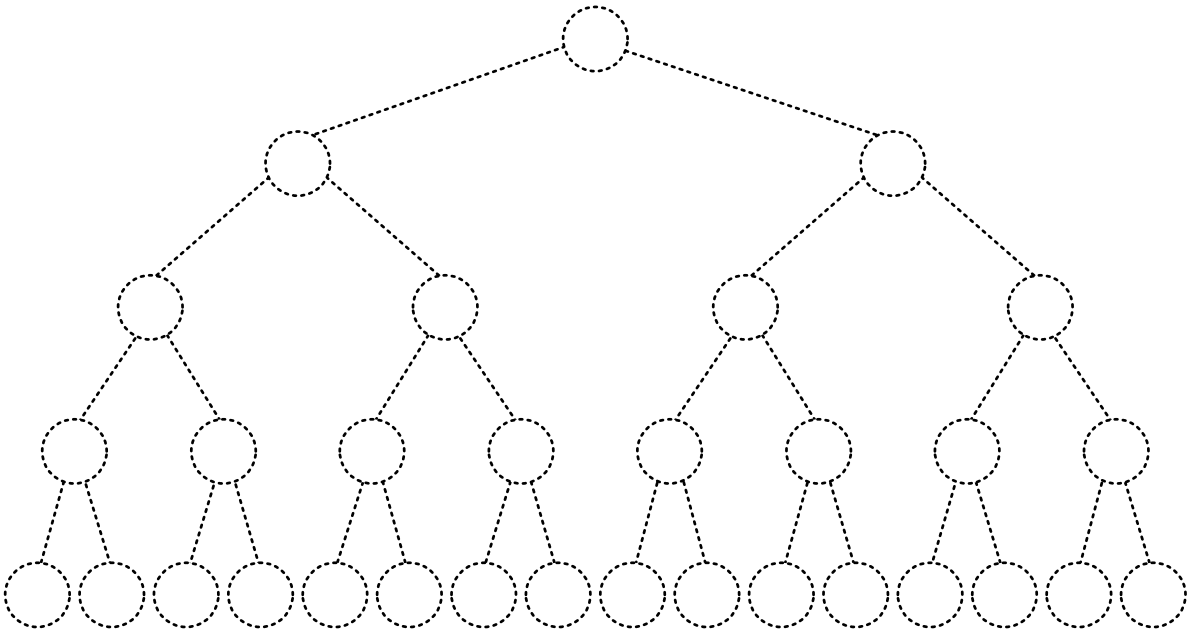
3.1) **(7 points)** Si l'affichage par niveaux de l'arbre binaire de recherche donne :

61, 35, 75, 12, 48, 69, 91, 5, 20

3.1.a) **(3 points)** Donnez la représentation graphique de l'arbre.

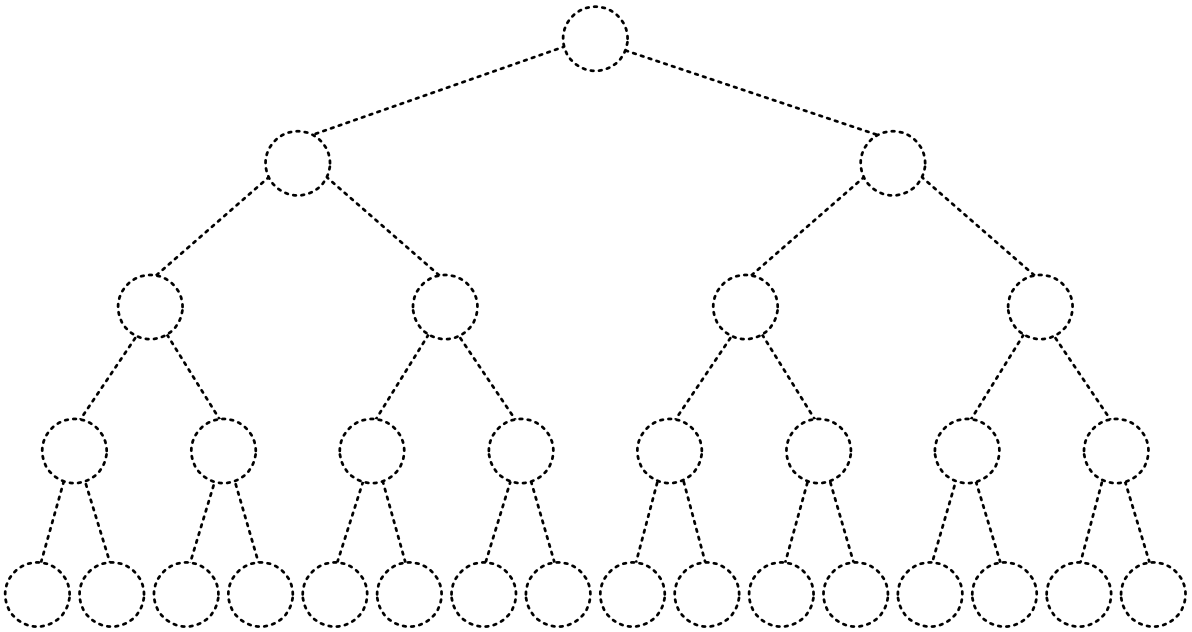


3.1.b) **(2 points)** Insérez un nœud contenant la clé 31 dans l'arbre binaire de recherche obtenu en 3.1.a**.**



3.1.c) **(2 points)** L'arbre obtenu en 3.1.a est-il un AVL ?

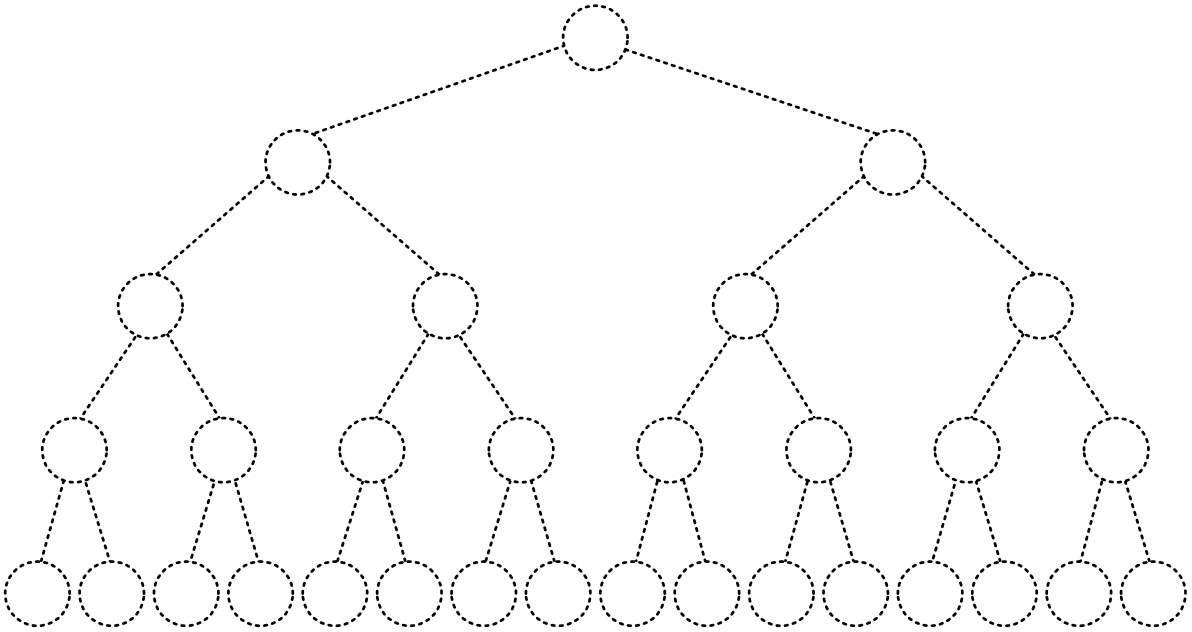
Si oui modifiez l'arbre obtenu en 3.1.b pour obtenir un AVL (insertion de type AVL de 31 dans l'arbre de 3.1.a). Sinon passez à la question suivante.



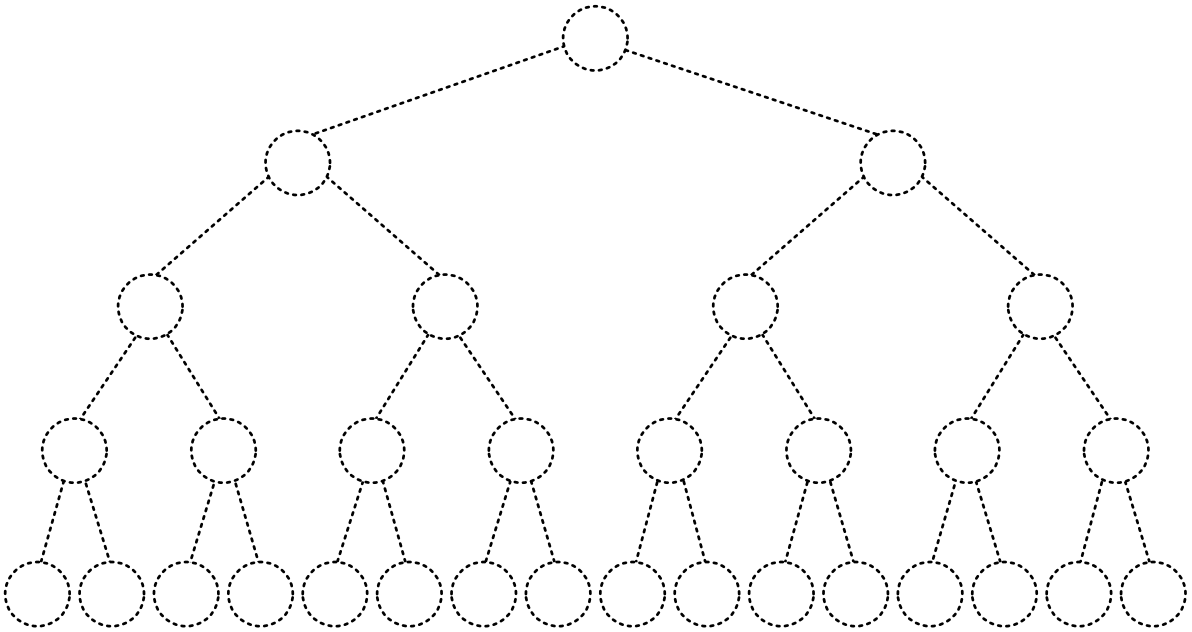
3.2) **(7 points)** Si l'affichage pré-ordre de l'arbre binaire de recherche donne :

61, 20, 12, 5, 48, 35, 75, 69, 91

3.2.a) **(3 points)** Donnez la représentation graphique de l'arbre.

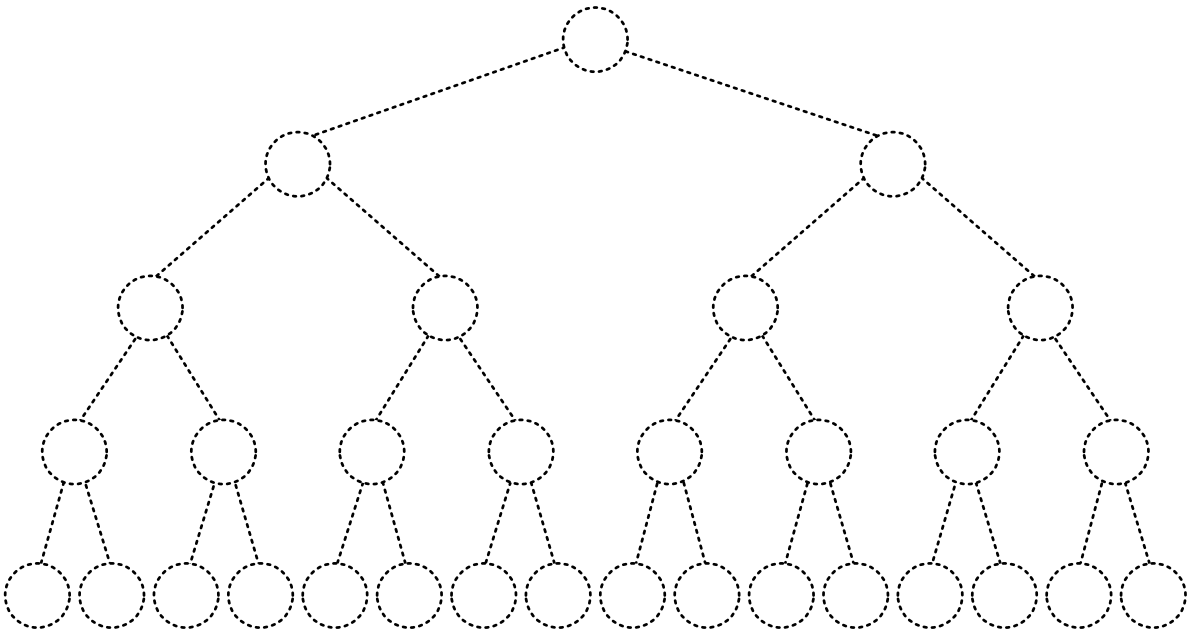


3.2.b) **(2 points)** Insérez un nœud contenant la clé 31 dans l'arbre binaire de recherche obtenu en 3.2.a**.**



3.2.c) **(2 points)** L'arbre obtenu en 3.2.a est-il un AVL ?

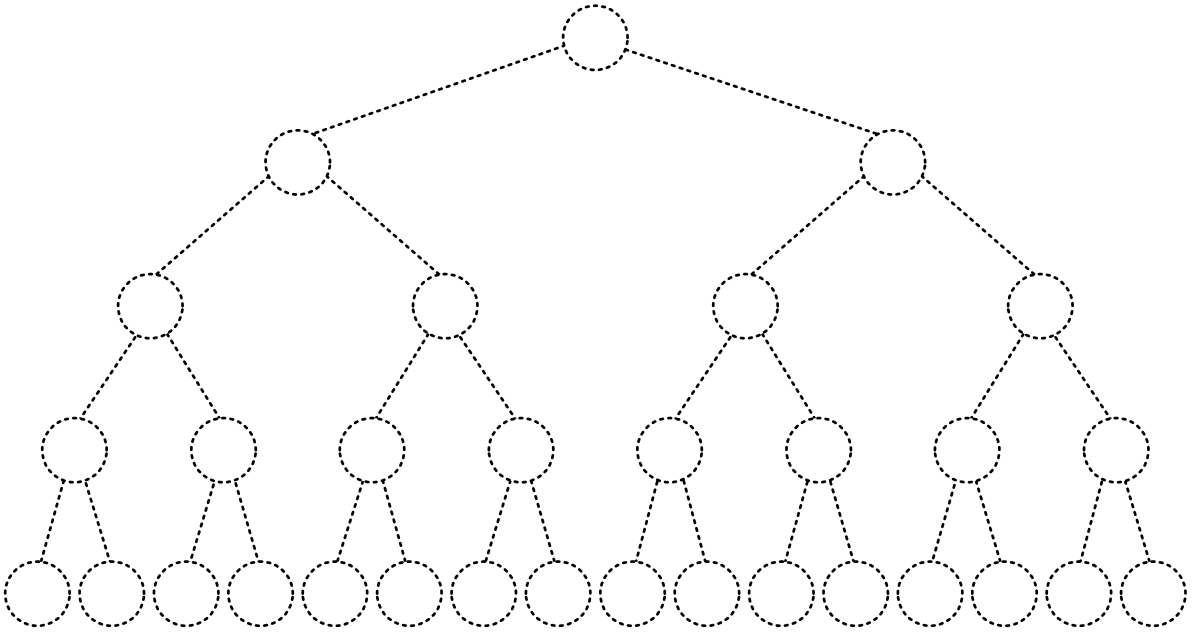
Si oui modifiez l'arbre obtenu en 3.2.b pour obtenir un AVL (insertion de type AVL de 31 dans l'arbre de 3.2.a). Sinon passez à la question suivante.



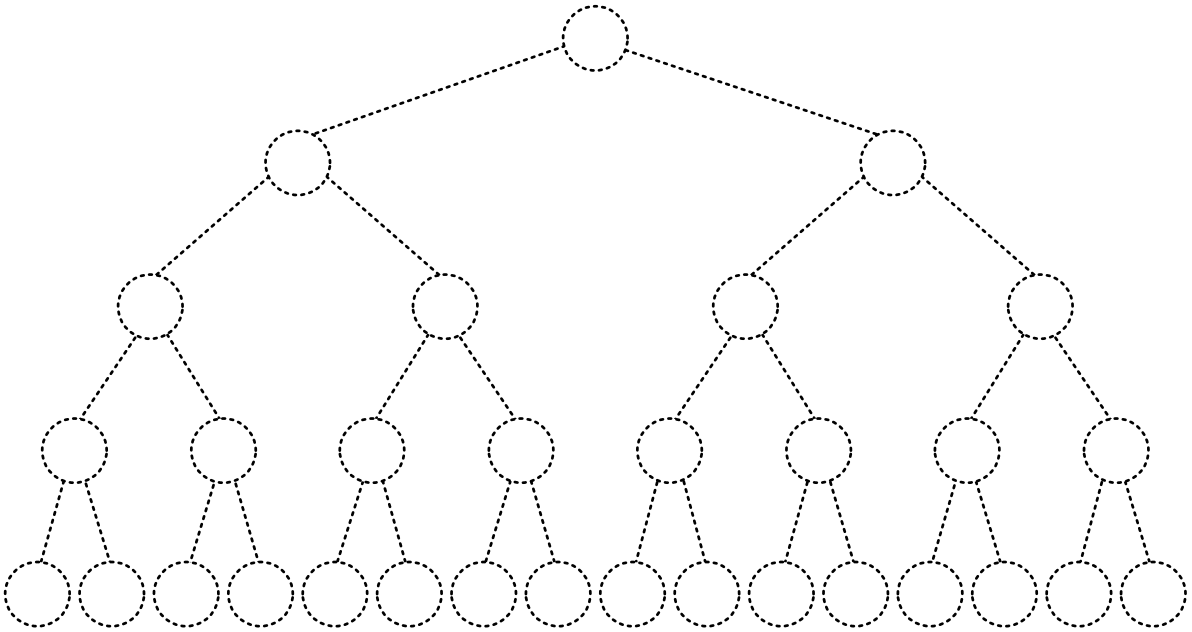
3.3) **(7 points)** Si l'affichage post-ordre de l'arbre binaire de recherche donne :

5, 20, 48, 35, 12, 69, 91, 75, 61

3.3.a) **(3 points)** Donnez la représentation graphique de l'arbre.

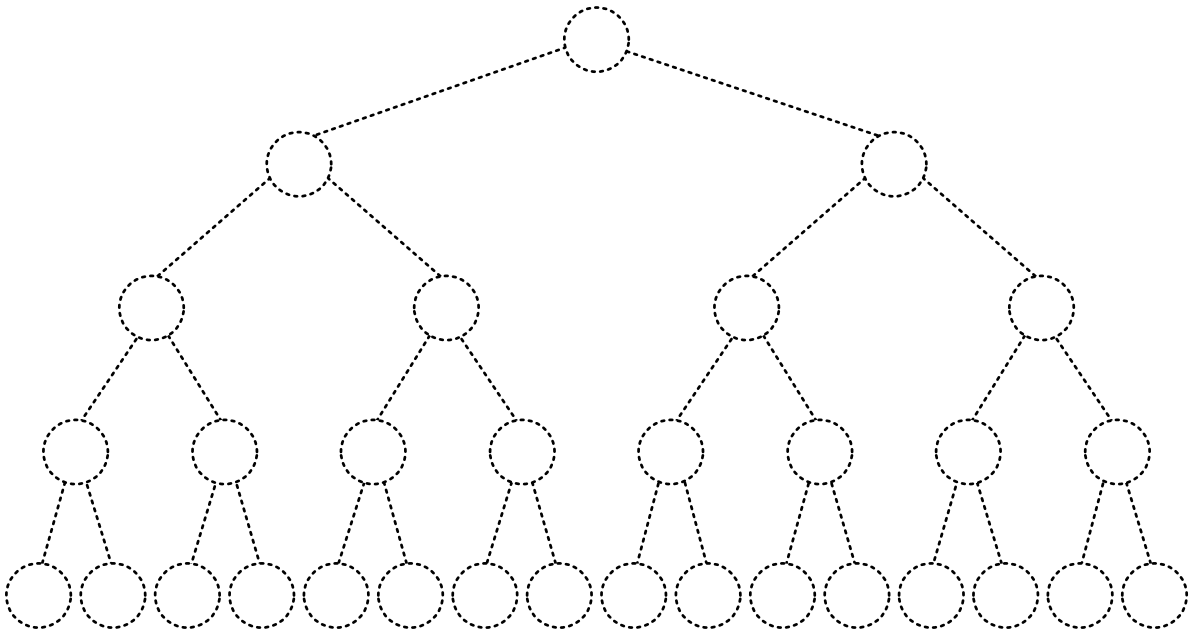


3.3.b) **(2 points)** Insérez un nœud contenant la clé 31 dans l'arbre binaire de recherche obtenu en 3.3.a**.**



3.3.c) **(2 points)** L'arbre obtenu en 3.3.a est-il un AVL ?

Si oui modifiez l'arbre obtenu en 3.3.b pour obtenir un AVL (insertion de type AVL de 31 dans l'arbre de 3.3.a). Sinon passez à la question suivante.

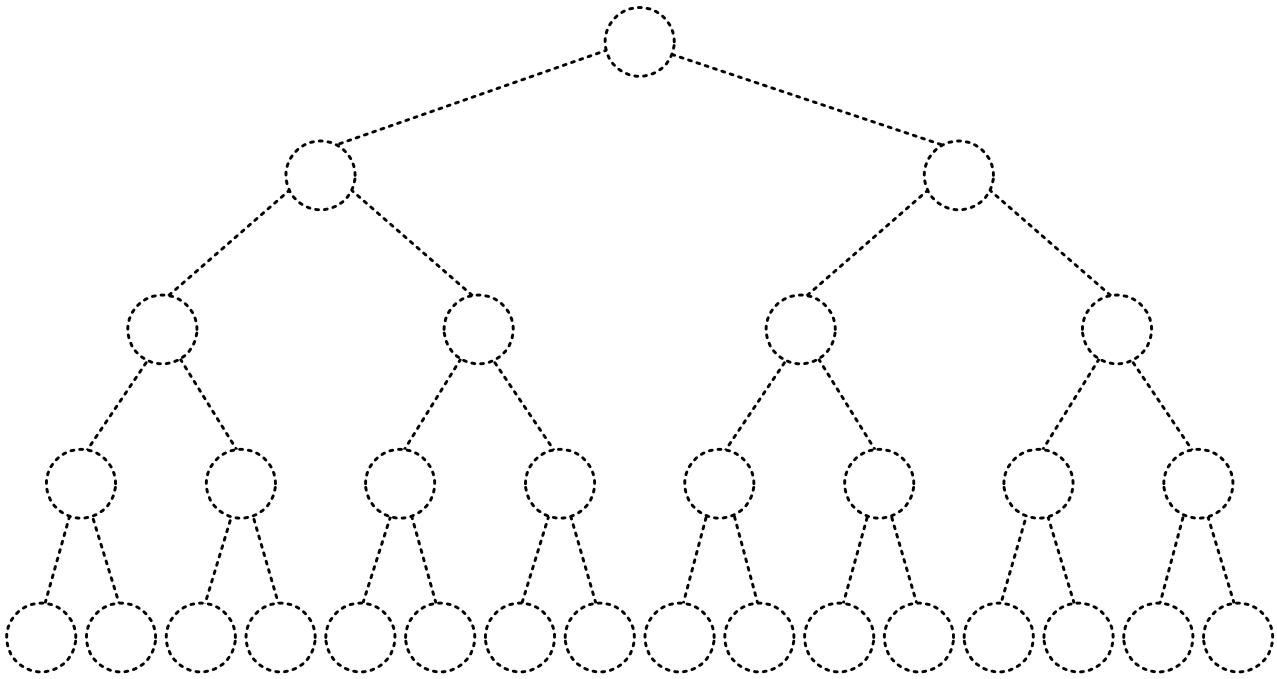


3.4) **(5 points)** L'affichage en ordre d'un arbre binaire de recherche ne permet pas d'en déduire la constitution. Néanmoins, sachant que :

- L'arbre binaire de recherche considéré est un AVL ;
- Si on effectuait une rotation simple vers la gauche auteur de la racine de cet arbre, l'arbre obtenu serait un arbre binaire complet ;
- Effectuer une rotation sur un nœud d'un arbre binaire de recherche produit un arbre binaire de recherche ;

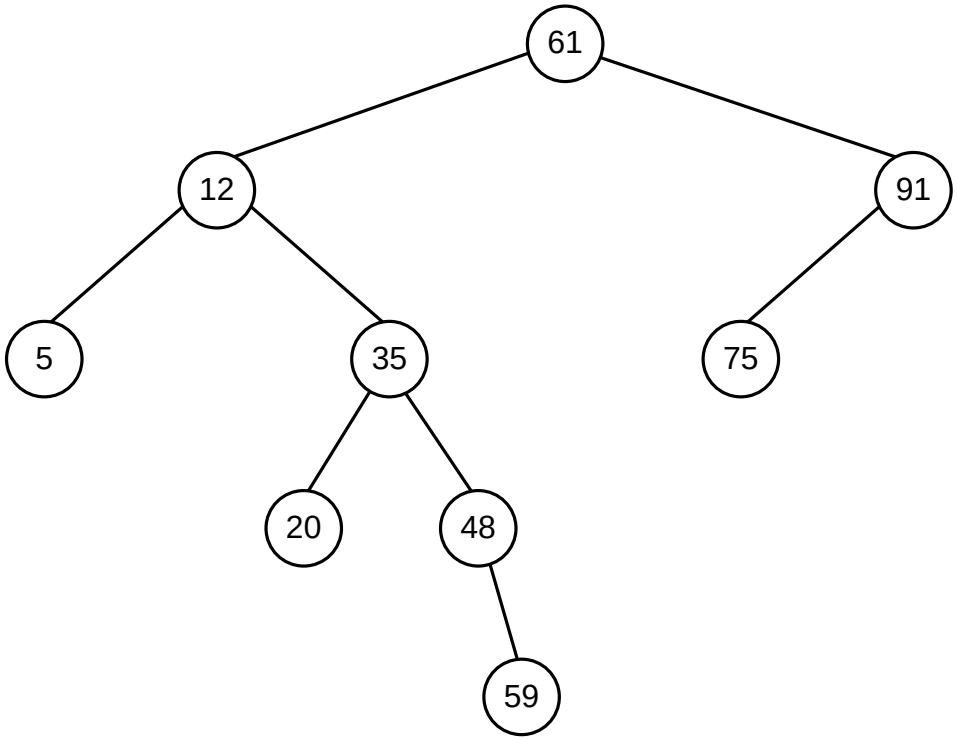
Donnez la représentation graphique de cet arbre AVL dont l'affiche en-ordre est :

3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33

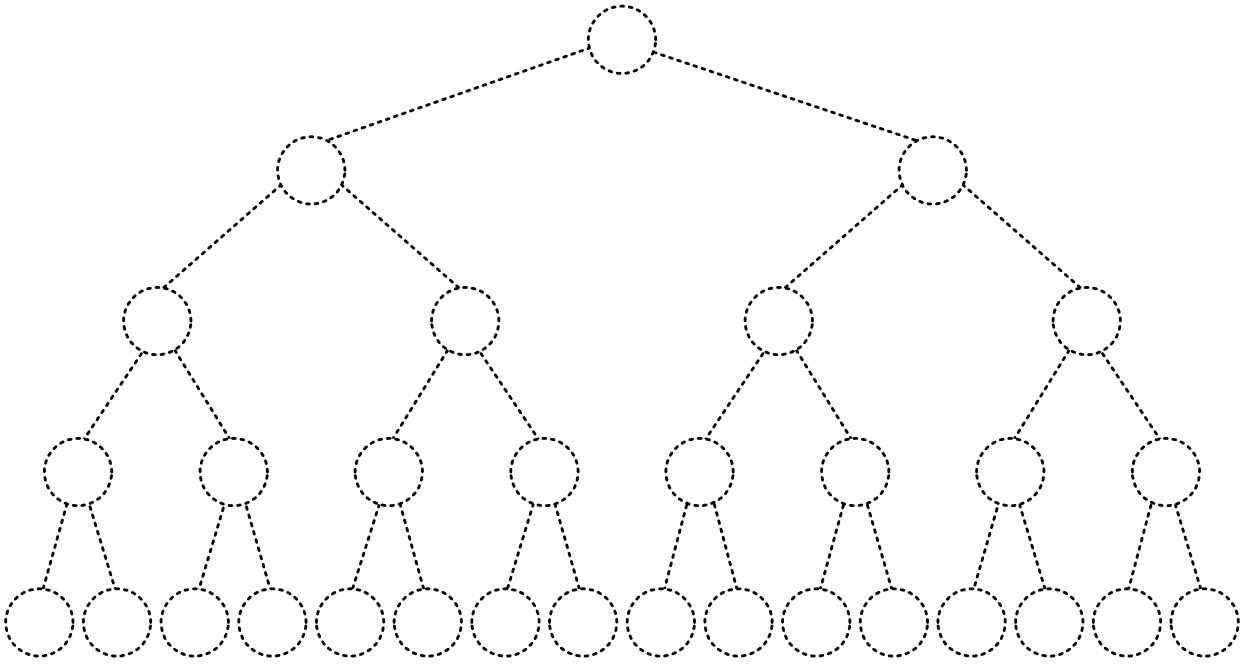


Question 4 : Arbre binaire de recherche de type Splay (16 points)

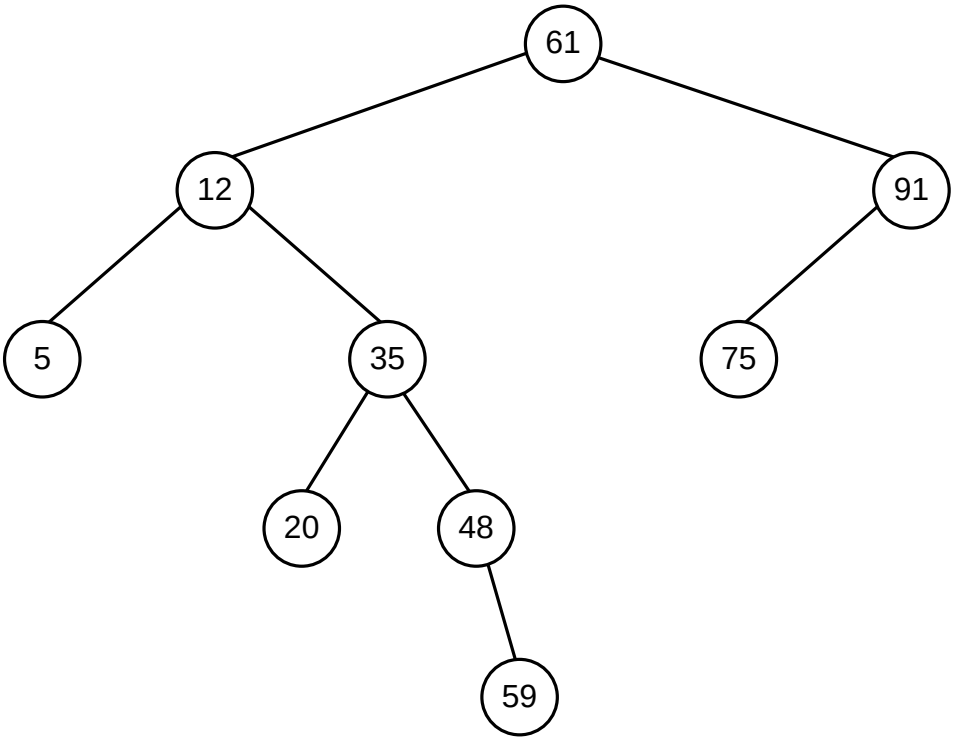
En partant de l'arbre Splay suivant :



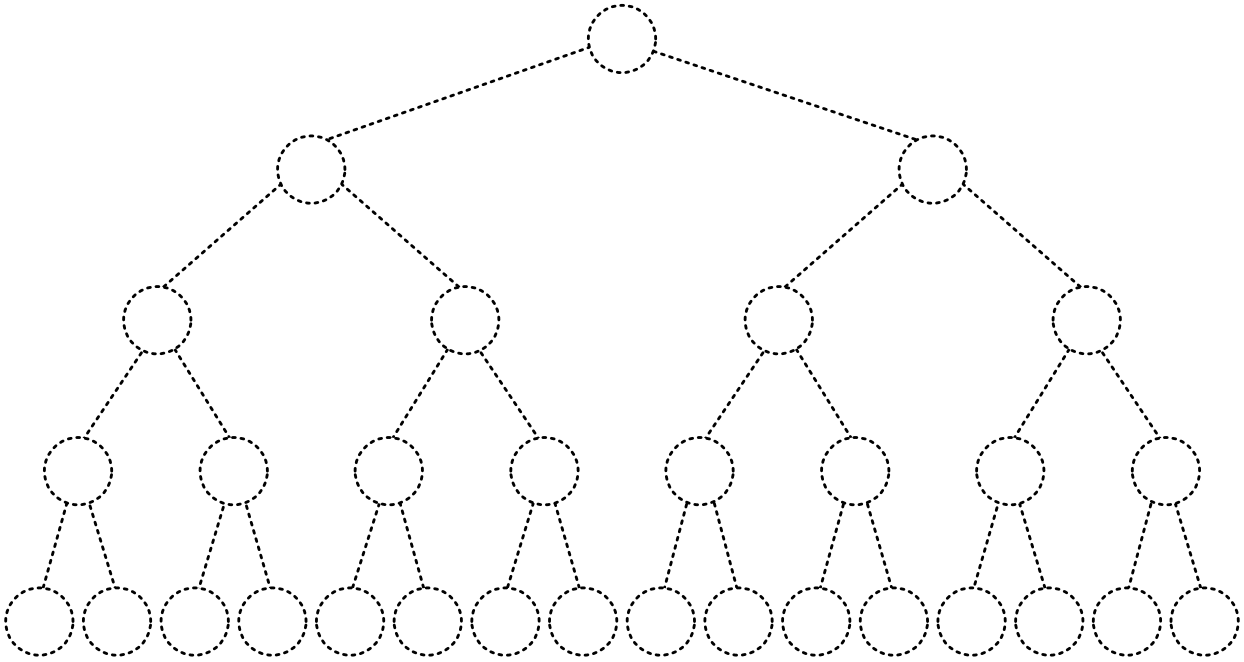
4.1) **(5 points)** Effectuez un `get(48)`.



En repartant du même arbre Splay :

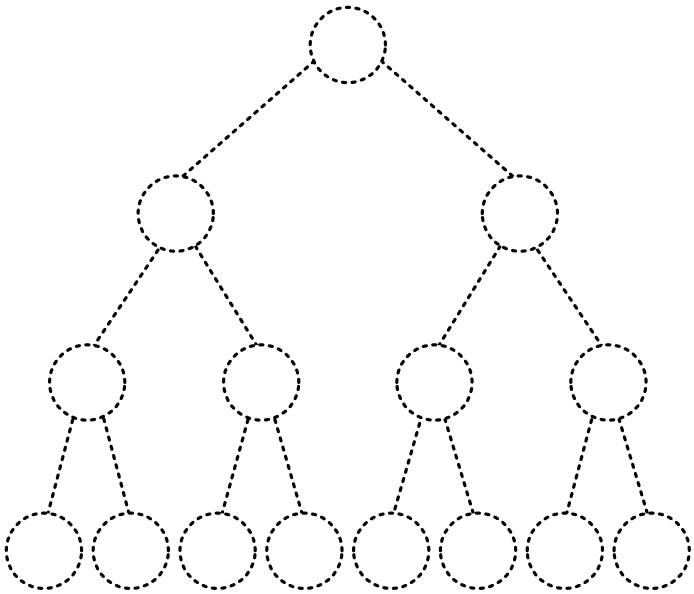


4.2) **(6 points)** Effectuez un `delete(59)`.

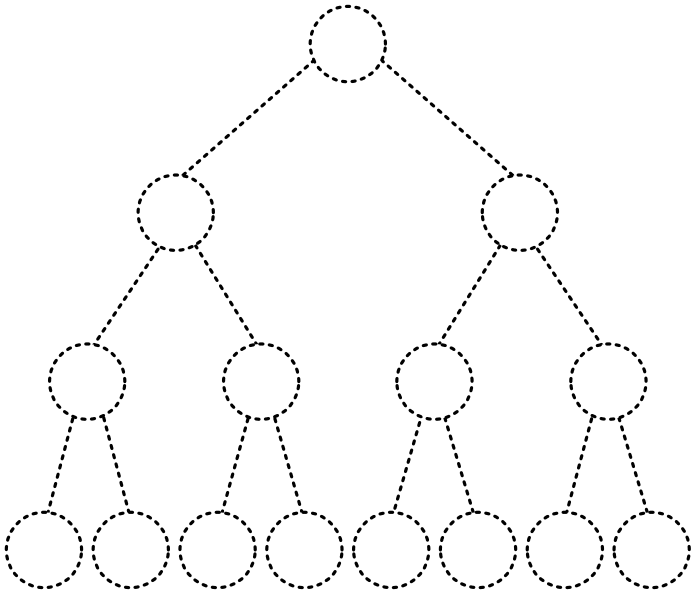


4.3) **(4 points)** Si le `get(48)` de la question 4.1) avait été exécuté par une implémentation de type top-down, quels auraient été les sous-arbres R et L juste avant que 48 ne soit placé à la racine ? **Aidez-vous des données de l'Annexe 3.**

Sous-arbre L:

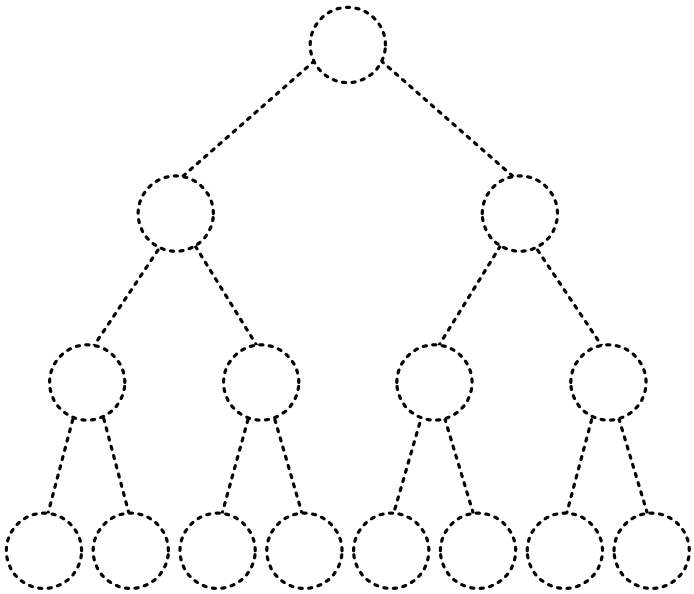


Sous-arbre R:

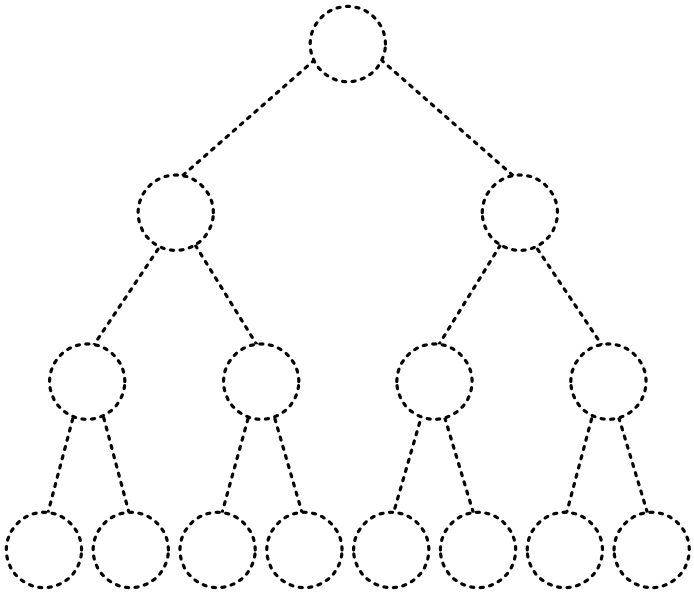


4.4) **(4 points)** Si le `delete(59)` de la question 4.2) avait été exécuté par une implémentation de type top-down, quels auraient été les sous-arbres R et L juste avant que 59 ne soit placé à la racine ? **Aidez-vous des données de l'Annexe 3.**

Sous-arbre L:



Sous-arbre R:



Question 5 : Généralités (19 points)

Répondez aux assertions suivantes par « vrai » ou par « faux » en justifiant votre réponse.

5.1) **(2 points)** La signature suivante est tout à fait correcte pour implémenter un itérateur sur la liste `MaListe`.

```
public class MaListe<T> implements Iterable<T>
{
    private int theSize;
    private T[] theItems;
        ...
    public java.util.Iterator<T> iterator( )
    { return new MonIterateur<T>( this ); }

    public class MonIterateur implements java.util.Iterator<T>
    {
        ...
    }
}
```

5.2) **(2 points)** Il est préférable d'utiliser une liste par tableau pour implémenter une FILE.

5.3) **(2 points)** Il est préférable d'utiliser une liste par tableau pour implémenter une PILE.

5.4) **(2 points)** Il est toujours possible d'insérer un élément dans une table de dispersement utilisant une résolution de collision par sondage quadratique dont la taille est un nombre premier.

5.5) **(2 points)** L'algorithme QuickSort a une complexité O($n \log(n)$) en pire cas.

5.6) **(2 points)** L'algorithme MergeSorte a une complexité O($n \log(n)$) en pire cas.

5.7) **(3 points)** Il est possible d'avoir un arbre AVL de hauteur h=10 possédant 120 nœuds.

5.8) **(4 points)** Construire un arbre binaire de recherche de sorte que le sous-arbre à la gauche de la racine est un AVL de 25 nœuds et que le sous-arbre à la droite de la racine est un AVL de 31 nœuds ne donne pas forcément un AVL.

Annexe 1

```
public class MyQuadraticProbingHashTable<AnyType>
{
    /** Constructeur par défaut
      */
    public MyQuadraticProbingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /** Constructeur avec paramètre
      */
    public QuadraticProbingHashTable( int size )
    {
        allocateArray( size );  makeEmpty( );
    }

    /** Insert x dans la table
      */
    public void insert( AnyType x )
    {
        int currentPos = findPos( x );
        if( isActive( currentPos ) )  return;

        array[ currentPos ] = new MyHashEntry<AnyType>( x, true );

        // Rehash
        if( ++currentSize >= array.length / 2 )  rehash( );
    }

    /** Augmente la taille de la table
      */
    private void rehash( )
    {
        MyHashEntry<AnyType> [ ] oldArray = array;

        allocateArray( nextPrime( 2 * oldArray.length ) );
        currentSize = 0;

        for( int i = 0; i < oldArray.length; i++ )
            if( oldArray[ i ] != null && oldArray[ i ].isActive )
                insert( oldArray[ i ].element );
    }

    /** Trouver la position de x
      */
    private int findPos( AnyType x )
    {
        int offset =  ; // masqué pour l'exercice
        int currentPos = myhash( x );

        while( array[ currentPos ] != null &&
                !array[ currentPos ].element.equals( x ) )
        {
            currentPos += offset;  // Compute ith probe
            offset +=  ; // masqué pour l'exercice
            if( currentPos >= array.length )
                currentPos -= array.length;
        }

        return currentPos;
    }
```

```
    /** Retire x
      */
    public void remove( AnyType x )
    {
        int currentPos = findPos( x );
        if( isActive( currentPos ) )
            array[ currentPos ].isActive = false;
    }

    /** Vérifie si x est contenu
      */
    public boolean contains( AnyType x )
    {
        int currentPos = findPos( x );
        return isActive( currentPos );
    }

    /** Vérifie si la case est active
      */
    private boolean isActive( int currentPos ) {
        return array[ currentPos ] != null && array[ currentPos ].isActive;
    }

    /** Vide la table
      */
    public void makeEmpty( ) {
        currentSize = 0;
        for( int i = 0; i < array.length; i++ )
            array[ i ] = null;
    }

    /** Donne le hash de x modulo taille de la table
      */
    private int myhash( AnyType x ) {
        int hashVal = x.hashCode( ) + 1;

        hashVal %= array.length;
        if( hashVal < 0 )
            hashVal += array.length;

        return hashVal;
    }

    /** Classe interne pour les alvéoles
      */
    private static class MyHashEntry<AnyType>
    {
        public AnyType  element;   // the element
        public boolean isActive;  // false if marked deleted

        public MyHashEntry( AnyType e ) {
            this( e, true );
        }

        public MyHashEntry( AnyType e, boolean i ) {
            element  = e;  isActive = i;
        }
    }

    private static final int DEFAULT_TABLE_SIZE = 11;

    private MyHashEntry<AnyType> [ ] array; // Tableau des éléments
    private int currentSize;                // Nombre d'alvéoles occupées
```

```
    /** Alloue l'espace mémoire
      */
    @SuppressWarnings("unchecked")
    private void allocateArray( int arraySize )
    {
        array = new MyHashEntry[ nextPrime( arraySize ) ];
    }

    /** Trouve le prochain nombre premier
      */
    private static int nextPrime( int n )
    {
        if( n <= 0 )
            n = 3;

        if( n % 2 == 0 )
            n++;

        for( ; !isPrime( n ); n += 2 )  ;

        return n;
    }

    /** Vérifie si n est premier
      */
    private static boolean isPrime( int n )
    {
        if( n == 2 || n == 3 )
            return true;

        if( n == 1 || n % 2 == 0 )
            return false;

        for( int i = 3; i * i <= n; i += 2 )
            if( n % i == 0 )
                return false;

        return true;
    }
}
```

Annexe 2

```
public final class IntraSort
{
   /**
   * Mergesort
   */
   @SuppressWarnings("unchecked")
   public static <AnyType extends Comparable<? super AnyType>>
   void mergesort( AnyType [ ] a )
   {
      AnyType [ ] tmpArray = (AnyType[]) new Comparable[ a.length ];

      mergesort( a, tmpArray, 0, a.length - 1 );
   }

   /**
   * Appel interne à mergesort
   */
   private static <AnyType extends Comparable<? super AnyType>>
   void mergesort( AnyType [ ] a, AnyType [ ] tmpArray,
                   int left, int right )
   {
      if( left < right )
      {
         int center = ( left + right ) / 2;
         mergesort( a, tmpArray, left, center );
         mergesort( a, tmpArray, center + 1, right );
         merge( a, tmpArray, left, center + 1, right );
      }
   }

   /**
   * Fusionne deux vecteurs
   */
   private static <AnyType extends Comparable<? super AnyType>>
   void merge( AnyType [ ] a, AnyType [ ] tmpArray,
               int leftPos, int rightPos, int rightEnd )
   {
      int leftEnd = rightPos - 1;
      int tmpPos = leftPos;
      int numElements = rightEnd - leftPos + 1;

      // Main loop
      while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];
      else
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      while( leftPos <= leftEnd )    // Copy rest of first half
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];

      while( rightPos <= rightEnd )  // Copy rest of right half
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      // Copy tmpArray back
      for( int i = 0; i < numElements; i++, rightEnd-- )
         a[ rightEnd ] = tmpArray[ rightEnd ];
   }
```

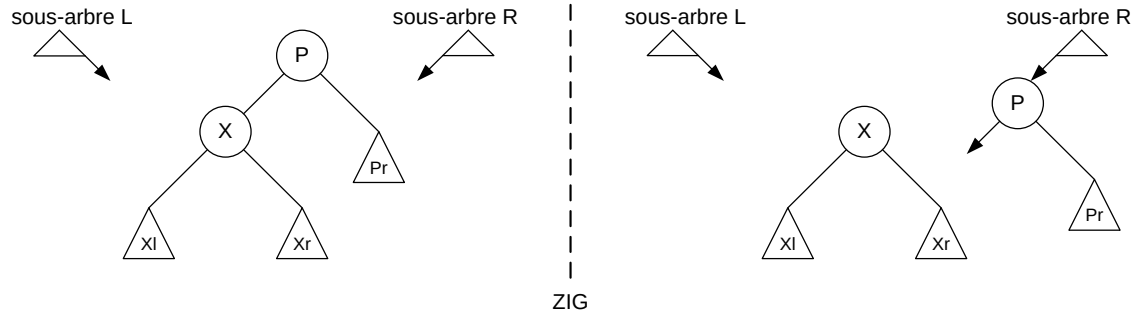
```
private static final int CUTOFF = 3;


/**
* Quicksort
*/
public static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a )
{
   quicksort( a, 0, a.length - 1 );
}

/**
* Appel interne à quicksort
* Utilise Median 3 et une valeur limite (cutoff) de 3.
*/
private static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a, int left, int right )
{
   if( left + CUTOFF <= right )
   {
      AnyType pivot = median3( a, left, right );

      // partitionnement
      int i = left, j = right - 1;
      for( ; ; )
      {
         while( a[ ++i ].compareTo( pivot ) < 0 ) { }
         while( a[ --j ].compareTo( pivot ) > 0 ) { }
         if( i < j )
         swapReferences( a, i, j );
         else
         break;
      }

      swapReferences( a, i, right - 1 );
      // fin du partitionnement

      // recursion
      quicksort( a, left, i - 1 );
      quicksort( a, i + 1, right );
   }
   else
   insertionSort( a, left, right );
}

/**
* Interchange (swap) deux valeurs
*/
public static <AnyType> void
swapReferences( AnyType [ ] a, int index1, int index2 )
{
   AnyType tmp = a[ index1 ];
   a[ index1 ] = a[ index2 ];
   a[ index2 ] = tmp;
}
```

```
    /**
    * Median 3
    */
    private static <AnyType extends Comparable<? super AnyType>>
    AnyType median3( AnyType [ ] a, int left, int right )
    {
       int center = ( left + right ) / 2;
       if( a[ center ].compareTo( a[ left ] ) < 0 )
          swapReferences( a, left, center );
       if( a[ right ].compareTo( a[ left ] ) < 0 )
          swapReferences( a, left, right );
       if( a[ right ].compareTo( a[ center ] ) < 0 )
          swapReferences( a, center, right );

       swapReferences( a, center, right - 1 );
       return a[ right - 1 ];
    }

    /**
    * insertionSort interne.
    * Utilisé par by quicksort.
    */
    private static <AnyType extends Comparable<? super AnyType>>
    void insertionSort( AnyType [ ] a, int left, int right )
    {
       for( int p = left + 1; p <= right; p++ )
       {
          AnyType tmp = a[ p ]; int j;

          for( j = p; j > left && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
          a[ j ] = a[ j - 1 ];
          a[ j ] = tmp;
       }
    }

    public static void main( String [ ] args )
    {
       Integer [ ] a = {16, 14, 12, 10, 8, 6, 4, 2,
                        15, 13, 11,  9, 7, 5, 3, 1},
                   b = {16, 14, 12, 10, 8, 6, 4, 2,
                        15, 13, 11,  9, 7, 5, 3, 1};

       mergesort( a );
       quicksort( b );

       for(Integer valeur : a) System.out.print(valeur + " ");
       System.out.println();
       for(Integer valeur : b) System.out.print(valeur + " ");

    }
}
```

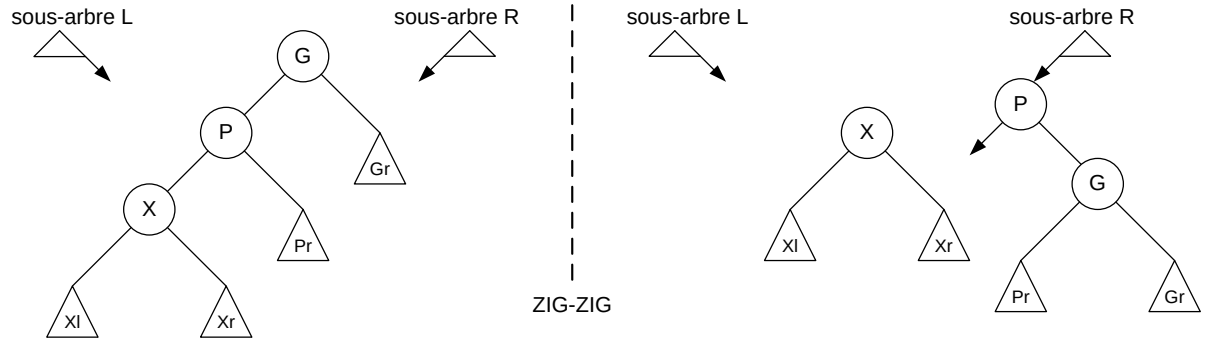
Annexe 3

Transformations TOP/DOWN utilisées dans les arbre Splay

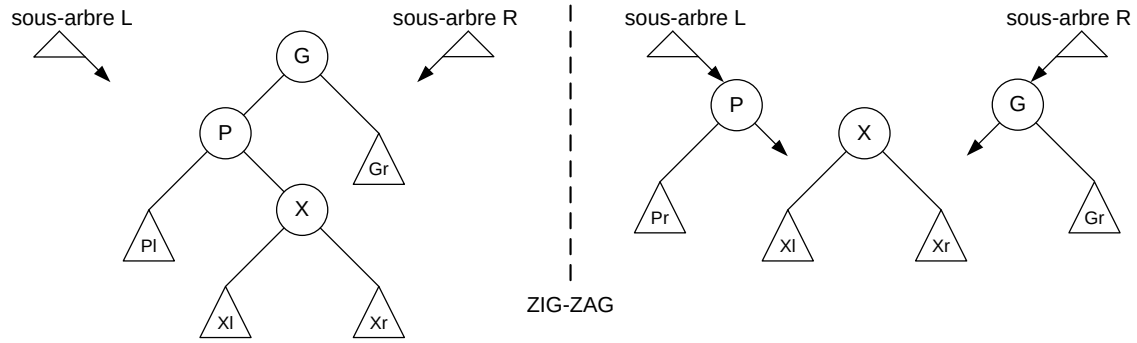
ZIG:



ZIG-ZIG



ZIG-ZAG



FIN: