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**	**20121**
Professeur		***Local***	***Téléphone***
Ettore Merlo, responsable – Tarek Ould Bachir, chargé de cours		**B-508**	**7128**
Jour	***Date***	***Durée***	***Heure***
Mercredi	**22 février 2009**	**2h00**	**13h00**

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

Directives particulières

Bonne chance à tous!

Important

Cet examen contient **6** questions sur un total de **18** 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 (20 points)

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

1.1) **(3 point)** Quelle est la complexité asymptotique en insertion de cette structure de données (considérez la complexité en cas moyen) ? Justifiez votre réponse.

1.2) **(6 points)** En vous servant du tableau ci-dessous, donnez l'état de la mémoire d'une table de taille N=11 après l'insertion des clés suivantes:

16, 23, 5, 36, 12.

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

1.3) **(6 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.

1.4) **(5 points)** Compléter la fonction `findPos()` donnée à l'Annexe 1 et reproduite ci-après. Pour mémoire, la fonction doit implémenter Hash(clé) = (clé + $2 \cdot i^2$) % 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;
}
```

Question 2 : Tris en $n \log(n)$ (20 points)

Exécuter l'algorithme « QuickSort » donné à l'Annexe 2 pour trier le vecteur suivant. La valeur *cut-off* pour l'algorithme « QuickSort » est 3.

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

Réponses :

2.1) (**4 points**) Donnez :

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

La valeur de la médiane (pivot) :

2.2) (**3 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																

2.3) (**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																

2.4) **(3 points)** La fonction récursive `quicksort` sera-telle rappelée après cette première récursion? Justifiez brièvement.

2.5) **(6 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 )
```

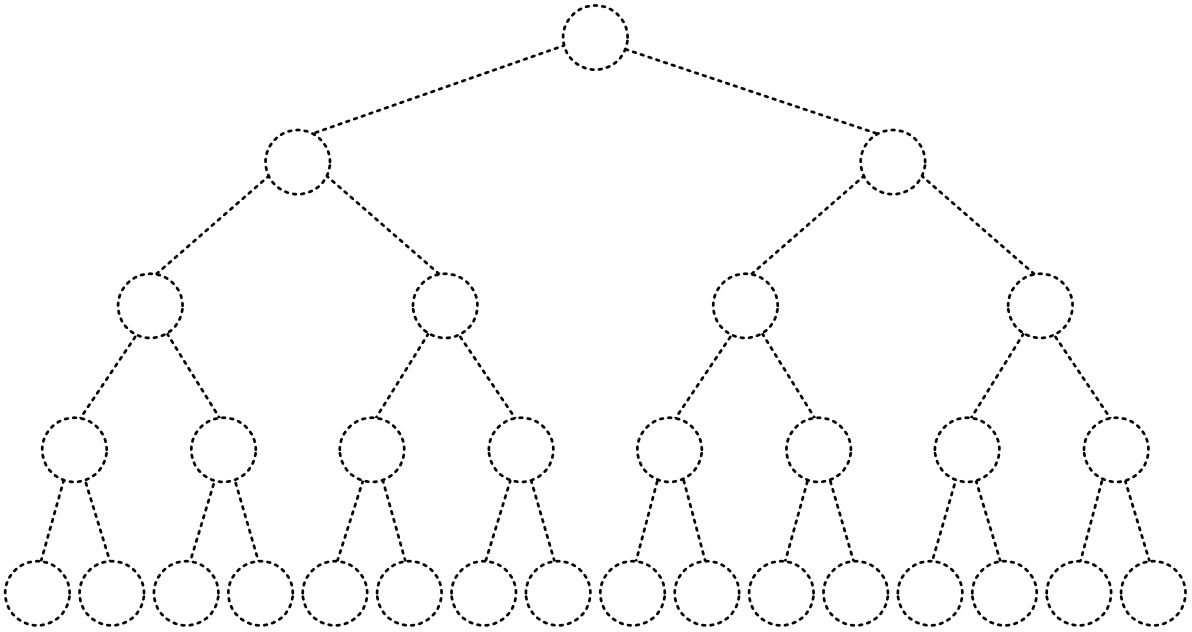
Votre réponse: _____

Question 3 : Arbres binaire de recherche (14 points)

Considérez les affichages des arbres binaires de recherche suivants. Pour chacun d'eux, donnez la représentation graphique de l'arbre. Lorsque demandé, dîtes si l'arbre binaire est un AVL.

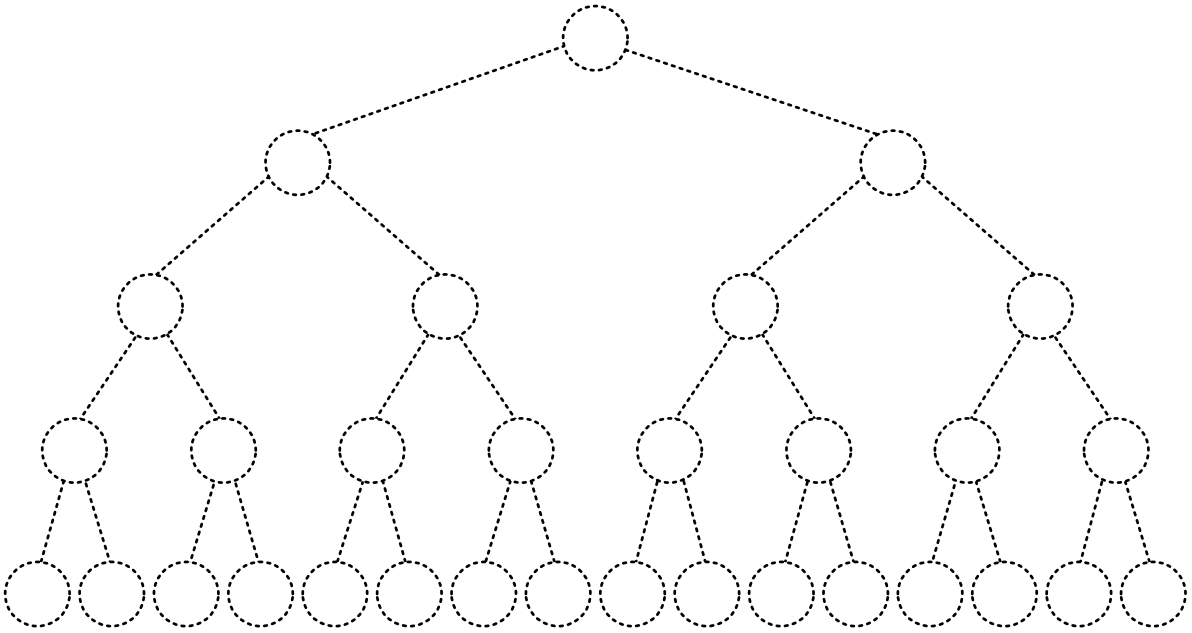
3.1) **(4 points)** L'affichage par niveaux de l'arbre binaire de recherche donne :

53, 35, 78, 43, 65, 87, 91, 88, 95 AVL? : _____



3.2) **(4 points)** L'affichage post-ordre de l'arbre binaire de recherche donne :

10, 4, 28, 23, 15, 35, 45, 29 AVL? : _____

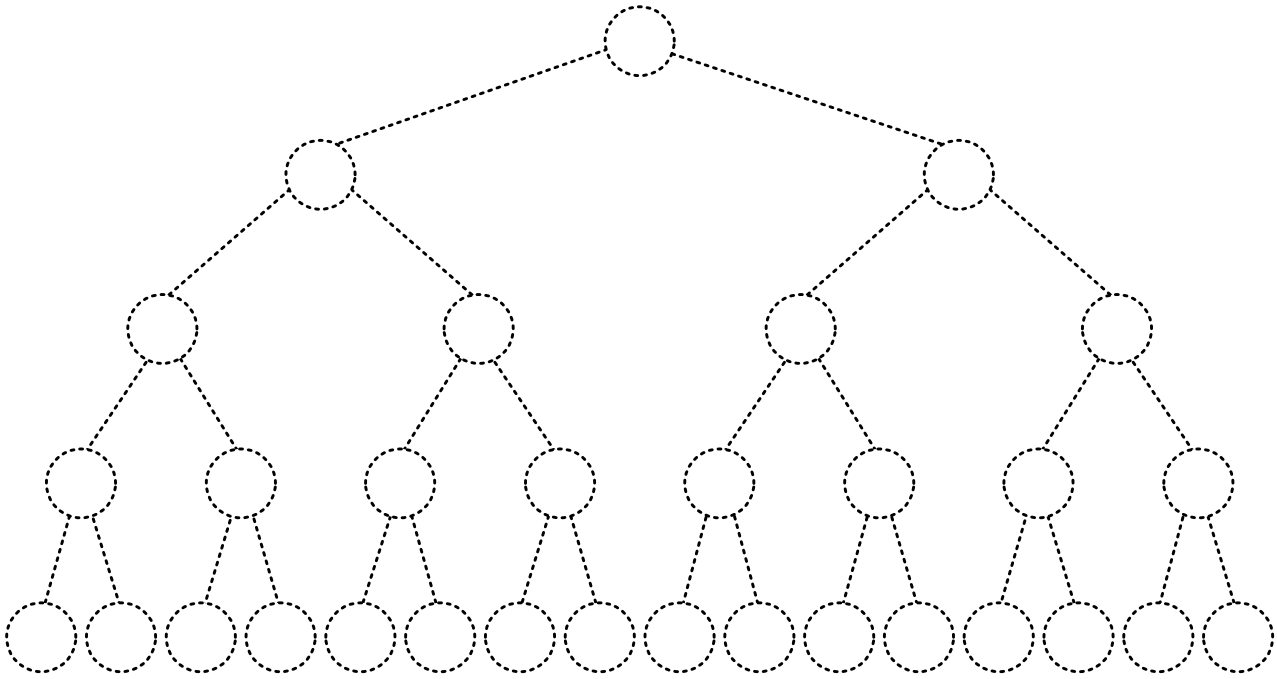


3.3) **(6 points)** L'affichage en ordre d'un arbre binaire de recherche ne permet pas d'en déduire la constitution. Néanmoins, sachant que :

- Dans le sous-arbre à la gauche de la racine, chaque nœud possède un sous arbre de gauche qui est plus haut que le sous-arbre de droite;
- Dans le sous-arbre à la droite de la racine, chaque nœud possède un sous arbre de droite qui est plus haut que le sous-arbre de gauche;
- L'arbre binaire de recherche considéré est un AVL;
- La fonction S(h) donnant le nombre minimal contenus dans un AVL respecte la relation $S(h) = S(h-1) + S(h-2) + 1$; $S(0) = 1$, $S(1) = 2$;

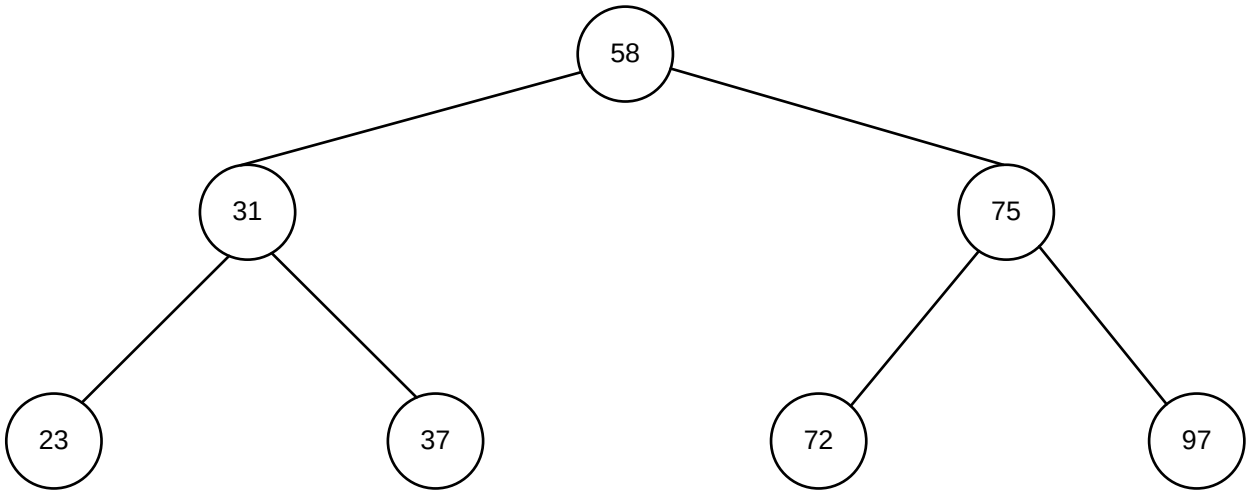
Donnez la représentation graphique de notre arbre AVL dont l'affiche en-ordre est :

2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30

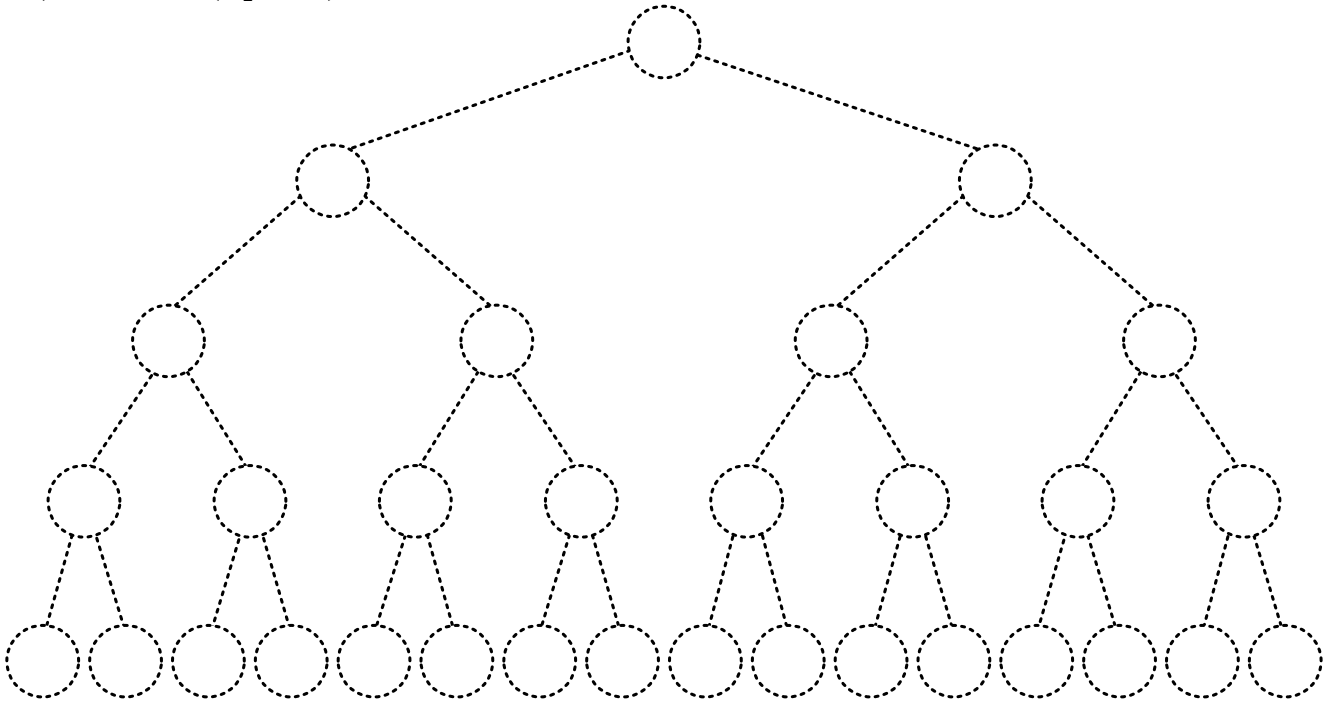


Question 4 : Arbre binaire de recherche de type AVL (16 points)

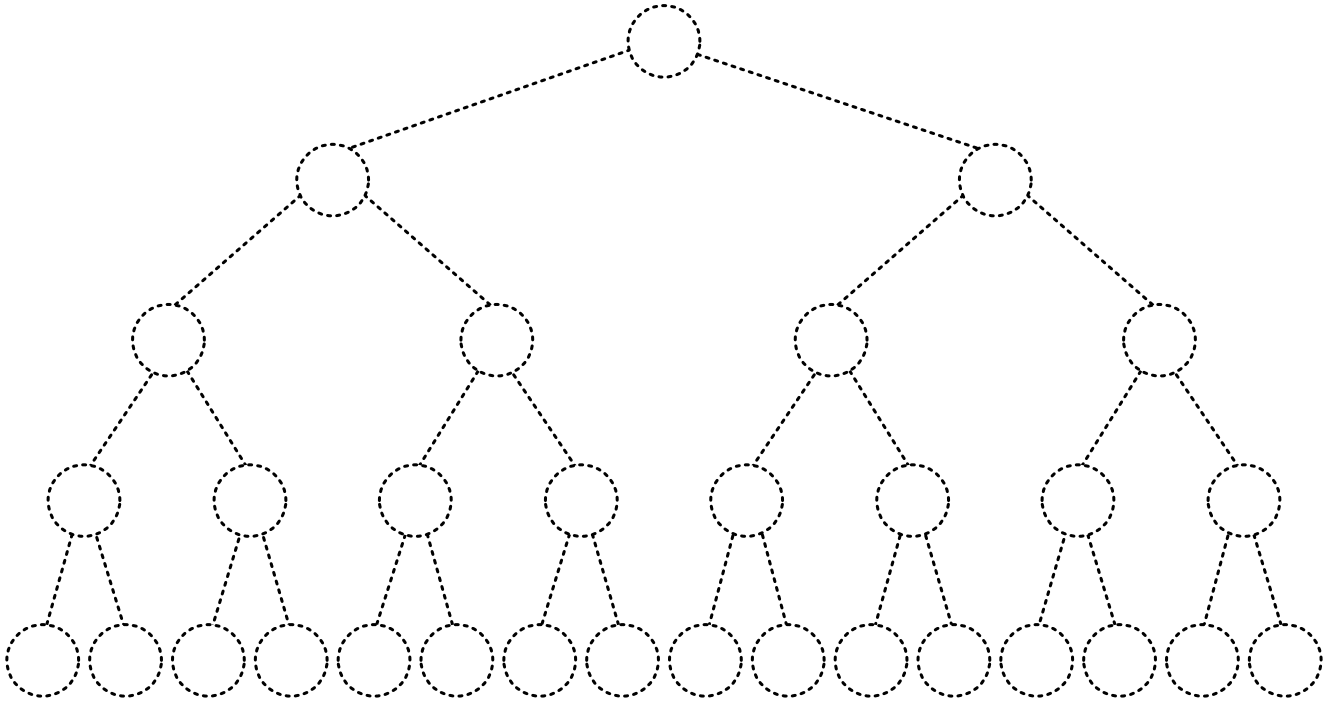
En partant de l'arbre AVL suivant :



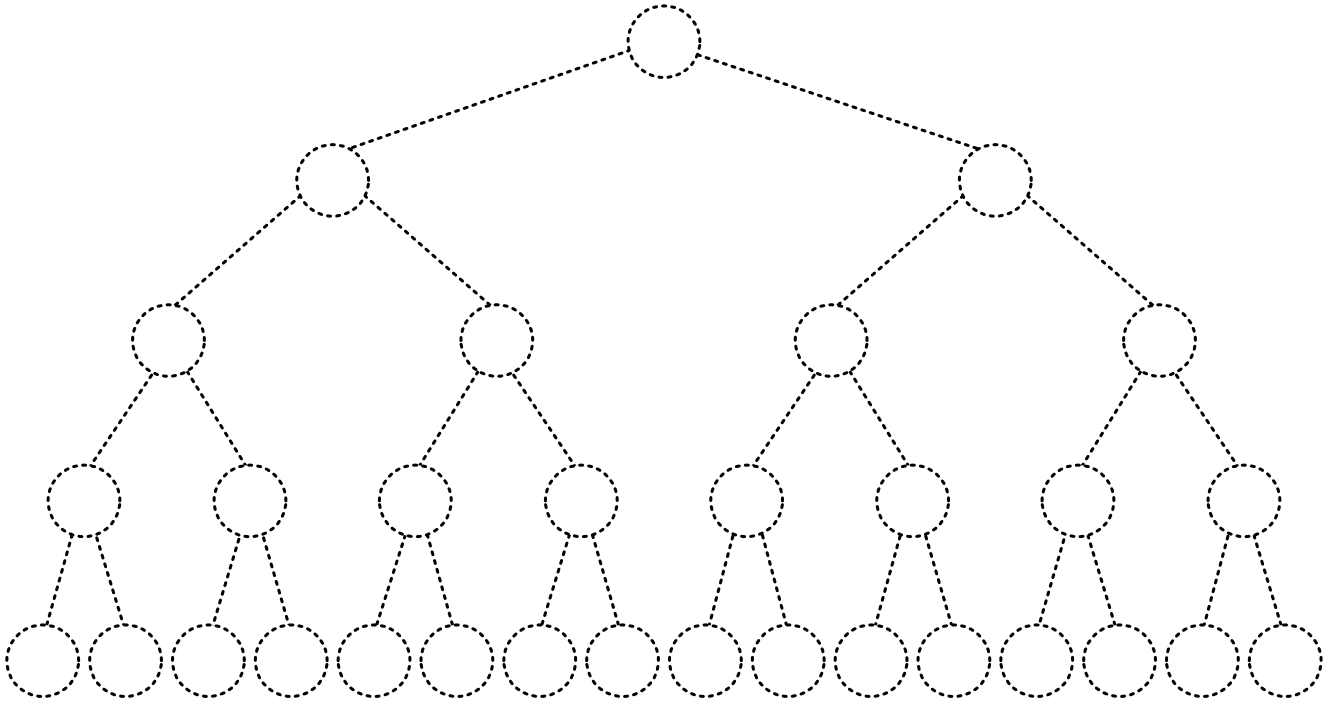
4.1) Insérez 41. **(2 points)**



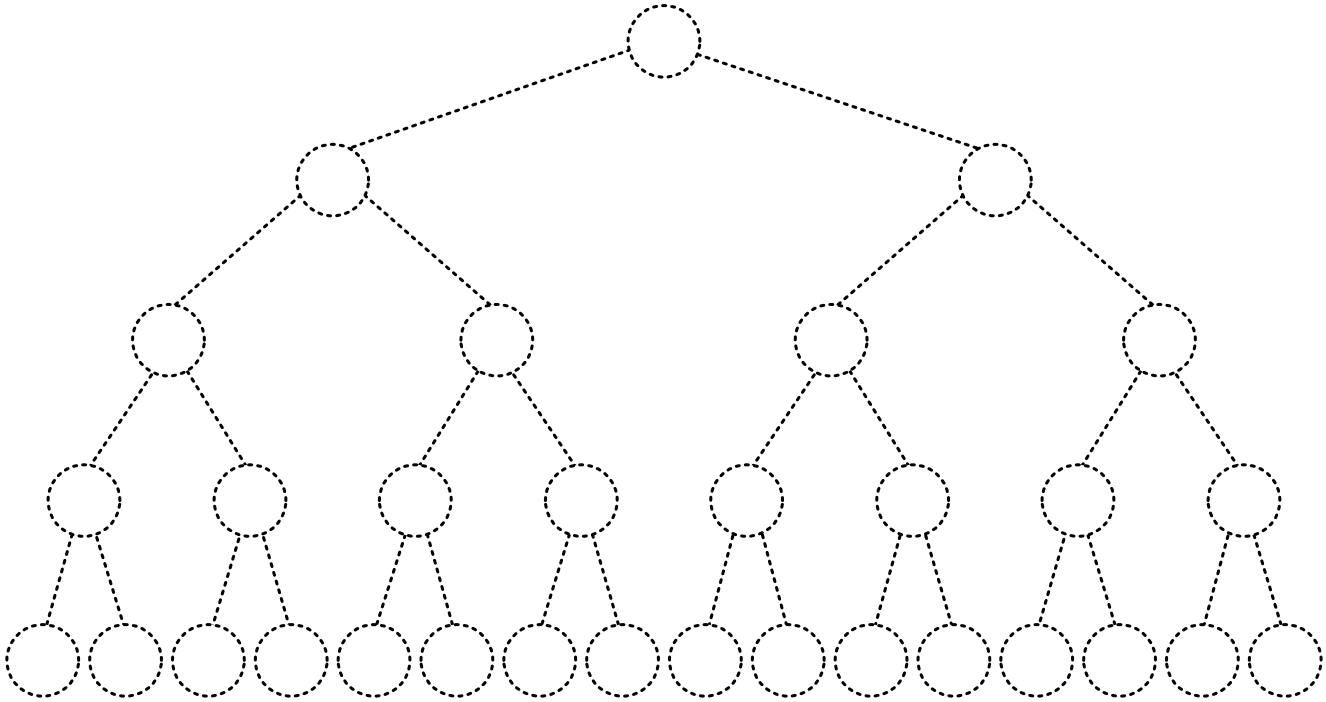
4.2) Insérez 49. **(2 points)**



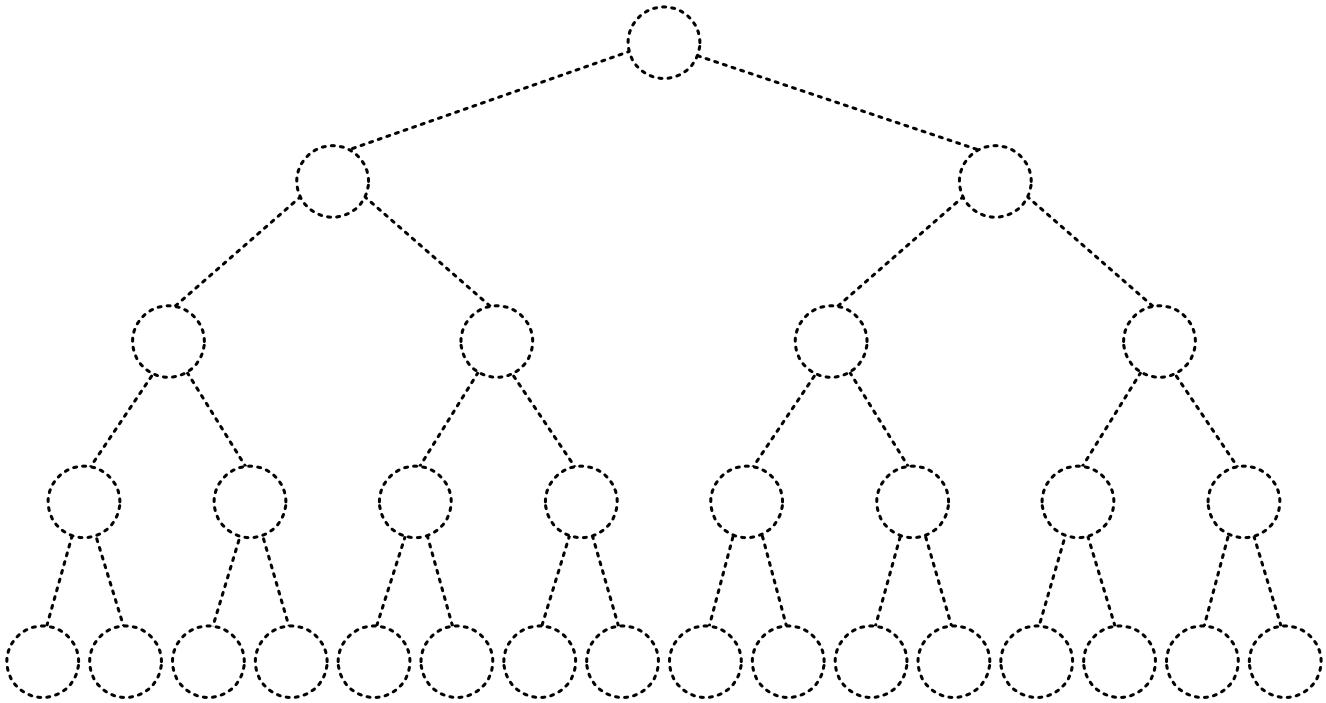
4.3) Insérez 69. **(2 points)**



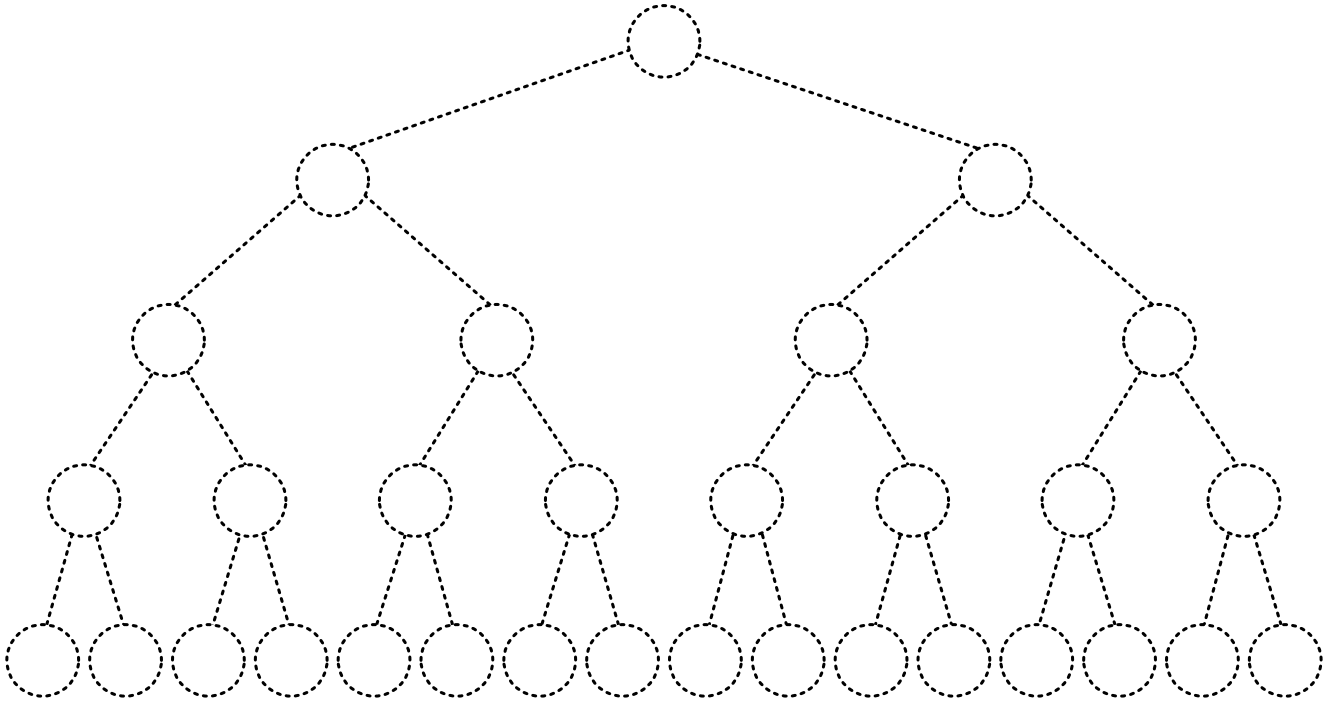
4.4) Insérez 60. **(2 points)**



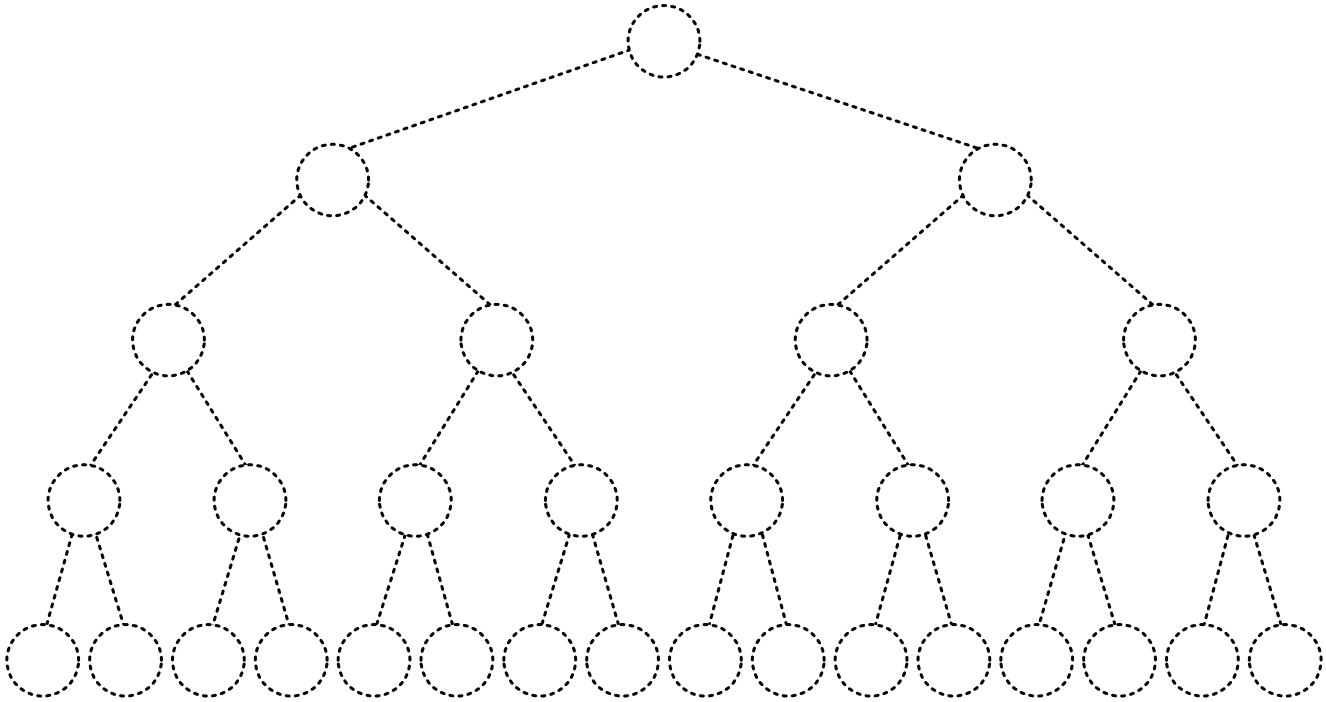
4.5) Insérez 29. **(2 points)**



4.6) Insérez 33. **(2 points)**

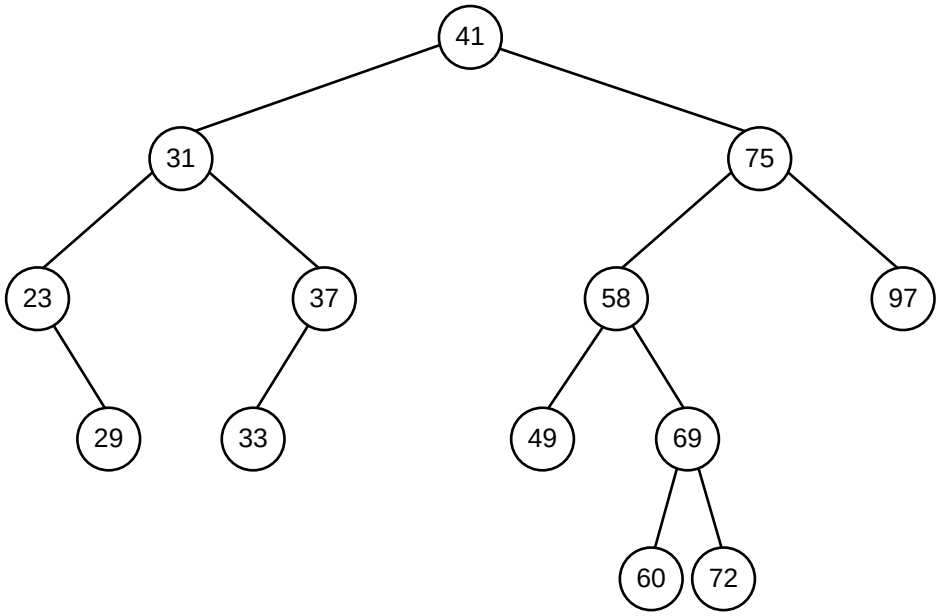


4.7) Insérez 32. **(4 points)**

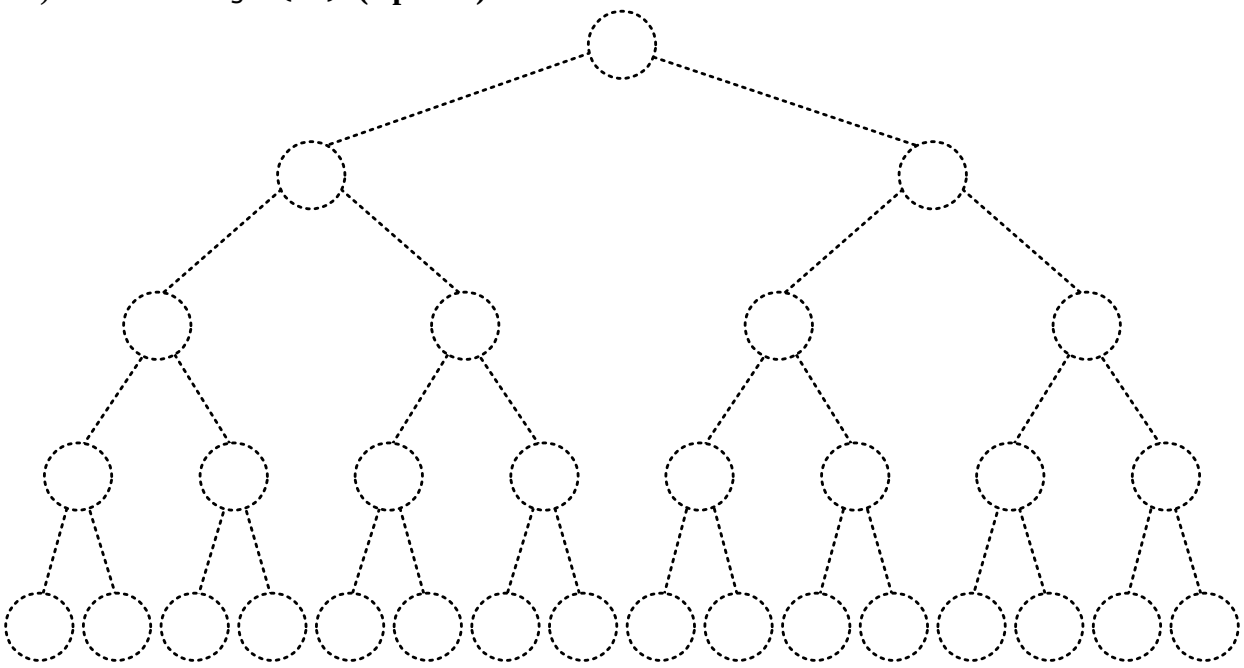


Question 5 : Arbre binaire de recherche de type Splay (20 points)

En partant de l'arbre Splay suivant :

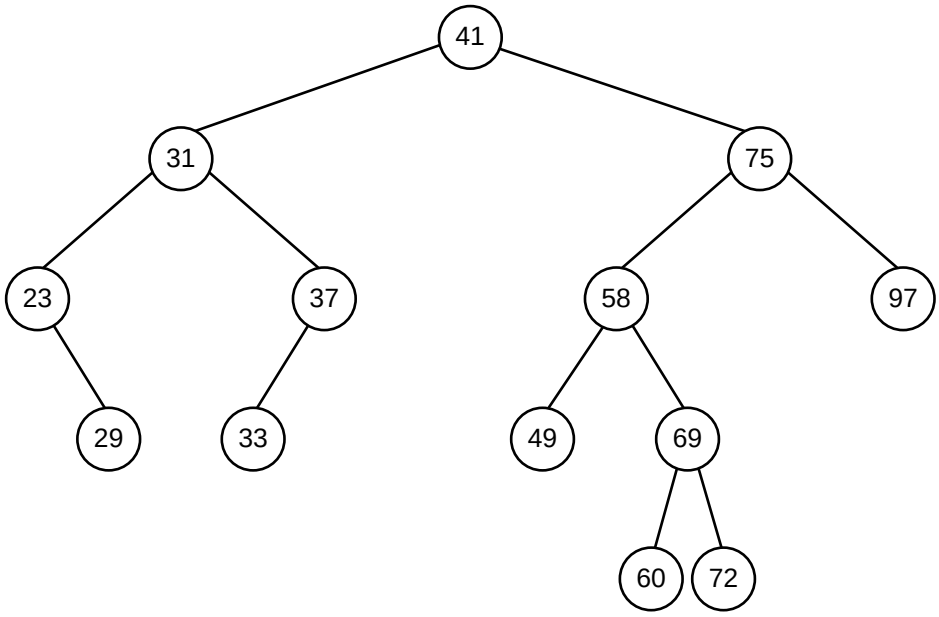


5.1) Effectuez un `get(72)`. **(6 points)**

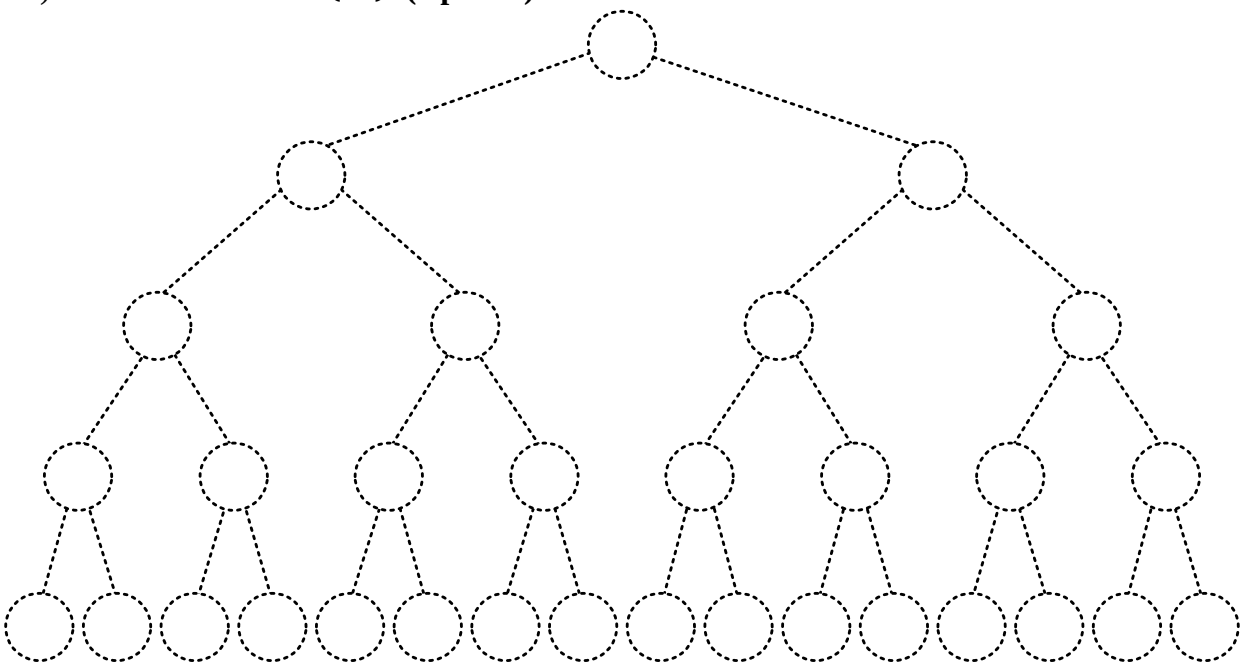


5.2) L'arbre Splay ainsi obtenu a-t-il une structure d'arbre AVL? **(4 points)**

En repartant du même arbre Splay :



5.3) Effectuez un `delete(69)`. **(6 points)**



5.4) L'arbre Splay ainsi obtenu a-t-il une structure d'arbre AVL? **(4 points)**

Question 6 : Généralités (10 points)

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

6.1) La complexité d'un algorithme de tri en cas moyen est au mieux O(n^2). **(2 points)**

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

6.3) Un `remove()` sur une liste chaînée s'effectue en O(1). **(2 points)**.

6.4) Un arbre AVL de hauteur h=5 possède au plus 20 nœuds. **(2 points)**.

6.5) Un arbre AVL ayant 54 nœuds a une hauteur d'au-plus h=7. **(2 points)**.

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; see Section 5.5
        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( );

        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 SortIntra
{

    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, 15, 14, 13, 12, 11, 10, 9,
                         8, 7, 6, 5, 4, 3, 2, 1};

        quicksort( a );

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

    }
}
```