ÉCOLE POLYTECHNIQUE MONTRÉAL

Le génie sans frontières

Questionnaire examen intra

INF2010

Sigle du cours

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

Sigle et titre du cours			*Groupe*	*Trimestre*
INF2010 – Structures de données et algorithmes			**Tous**	**20091**
Professeur			***Local***	***Téléphone***
Ettore Merlo, responsable – Tarek Ould Bachir, chargé de cours			**B-418**	**5758**
Jour	***Date***	***Durée***		***Heure***
Mercredi	**18 février 2009**	**2h00**		**9h00**

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 **5** questions sur un total de **17** 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 (16 points)

Considérez une table de dispersement par débordement progressif de dimension 11 avec "*sondage*" à double dispersement (f(i) = i * $h_2(x)$), avec fonction de dispersement primaire $h_1(x) = x \% 11$ et avec fonctions de dispersement secondaire $h_2(x) = 7 - (x \% 7)$.

1.1) **(12 pnt)** Quelles sont les valeurs i et $h_i(x)$ utilisées pour l'insertion dans l'ordre de chaque clef 36, 80, 16, 47. Détaillez les calculs.

i) 36

ii) 80

iii) 16

iv) 47

1.2) **(4 pnt)** Dessinez l'état de la table à la fin de l'insertion de toutes les clefs indiquées.

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

Question 2 : Tris en *n* log(*n*) (16 points)

Exécuter l'algorithme « MergeSort » et « QuickSort » pour trier le vecteur suivant avec un index de position qui commence à zéro. La valeur *cut-off* pour l'algorithme « QuickSort » est 10.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Valeurs	47	99	83	49	12	63	19	59	69	31	12	3	51	47	9	18

Réponse :

2.1) **(6 pnt)** « MergeSort » :

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Valeurs	47	99	83	49	12	63	19	59	69	31	12	3	51	47	9	18

2.2) **(10 pnt)** « QuickSort » :

a) **(6 pnt)** Partie « QuickSort »

Les trois valeurs de « Median3 » : Valeur médiane :

État du vecteur après l'exécution de Median 3

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

État du vecteur après l'exécution du premier round du tri QuickSort (partitionnement)

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

b) **(3 pnt)** Recours à « InsertionSort » après cut-off

État du vecteur à la gauche de la médiane après InsertionSort :

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

État du vecteur à la droite de la médiane après InsertionSort :

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

c) **(1 pnt)** Résultat final

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

Question 3 Tri quadratique (variation du *BubbleSort*) (20 points)

Considérer le code source Java donné à l'annexe 1 pour trier le vecteur qui suit :

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

3.1) **(10 pnt)** Si une boucle n'est pas exécutée, laisser la case vide

3.1.a)
```
While: st = 0 ; limit = 7
    For 1: j= 0 ... 6
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

```
    For 2: j= 6 ... 0
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

3.1.b)
```
While: st = 1 ; limit = 6
    For 1: j= 1 ... 5
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

```
    For 2: j= 5 ... 1
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

3.1.c)
```
While: st = 2 ; limit = 5
    For 1: j= 2 ... 4
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

```
For 2: j= 4 ... 2
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

3.1.d)
```
While: st = 3 ; limit = 4
For 1: j= 3 ... 3
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

```
For 2: j= 3 ... 3
```

État du vecteur après la boucle

Index	0	1	2	3	4	5	6	7
Valeurs								

3.2) **(4 pnt)** Donner une estimation de complexité algorithmique en pire et en meilleur cas. Justifier brièvement.

3.3) **(6 pnt)** Comparer le temps d'exécution de cet algorithme à celui de BubbleSort (plus lent, plus rapide). Argumenter la réponse.

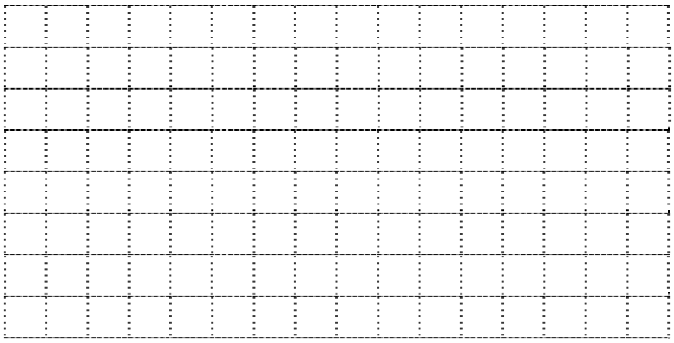
Question 4 : Arbre binaire de recherche (30 points)

Considérer le code source Java donné aux annexes 2 et 3 pour manipuler un arbre binaire de recherche :

4.1) **(2 pnt)** Donner le résultat du premier affichage (annexe 2) :

```
// Creation d'un arbre vide
BinarySearchTree<Integer> t = new BinarySearchTree<Integer>( );

/** Premier affichage: Etat de l'arbre*/
System.out.println( "Premier Arbre:" );
t.printTree( );
```

Utiliser la grille suivante pour ce faire ; chaque colonne représente un caractère :



4.2) **(4 pnt)** Donner dans l'ordre les valeurs **j** insérées dans l'arbre par l'appel (annexe 2) :

```
// Remplissage de l,arbre avec des valeurs
for(int i=0; i < PROFONDEUR_MAX; ++i)
        for(int j = ( 1 << (3-i) ); j < ( 1 << 4 ); j += ( 1 << (4-i) ) )
                t.insert( j );
```

Pour mémoire, l'opérateur << effectue un décalage à gauche, de sorte que :

```
int i=1;
i << 0;// 1
i << 1;// 2
i << 2;// 4
i << 3;// 8
// etc...
```

Dans l'ordre, les valeurs insérées sont :

4.3) **(3 pnt)** Donner dans l'ordre les valeurs **i** retirées de l'arbre par l'appel (annexe 2) :

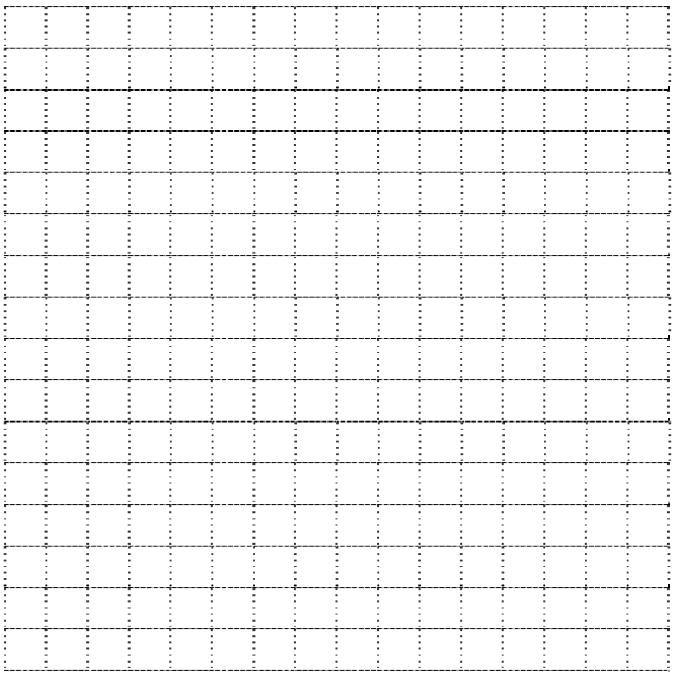
```
// Retirer certaines valeurs
for(int i =  (1 << PROFONDEUR_MAX) - 1; i > (1 << 3); i -= 1)
        t.remove( i );
```

Dans l'ordre, les valeurs retirées sont :

4.4) **(6 pnt)** Donner le résultat du deuxième affichage (annexe 2) :

```
/** Second affichage: Etat de l'arbre*/
System.out.println( "\nSecond Arbre:" );
// Remplissage de l'arbre avec des valeurs
// ... code 4.2)
t.printTree( );
```

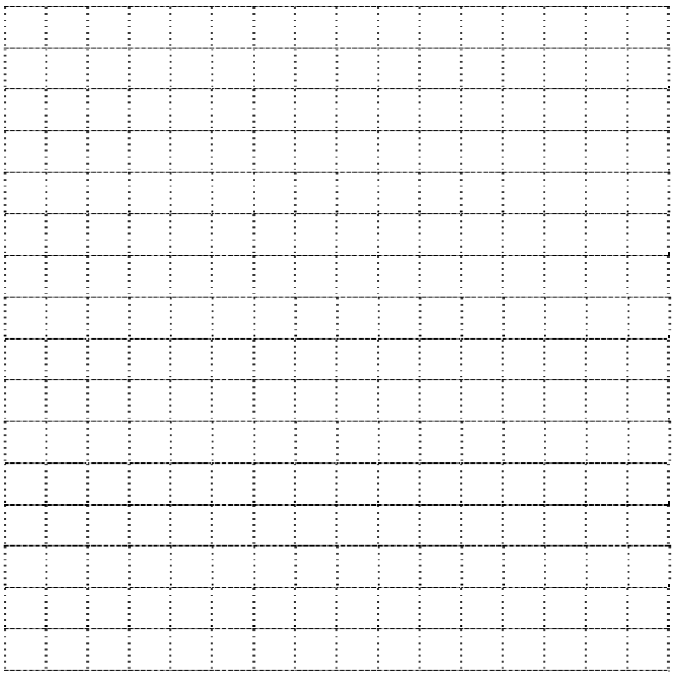
Utiliser la grille suivante pour ce faire ; chaque colonne représente un caractère :



4.5) **(6 pnt)** Donner le résultat du quatrième affichage (annexe 2) :

```
/** Quatrième affichage: Etat de l'arbre*/
System.out.println( "\nTroisième Arbre:" );
// Retirer certaines valeurs
// ... code 4.3)
t.printTree( );
```

Utiliser la grille suivante pour ce faire ; chaque colonne représente un caractère :



4.6) **(4 pnt)** Donner le résultat du troisième et cinquième affichage (annexe 2) :

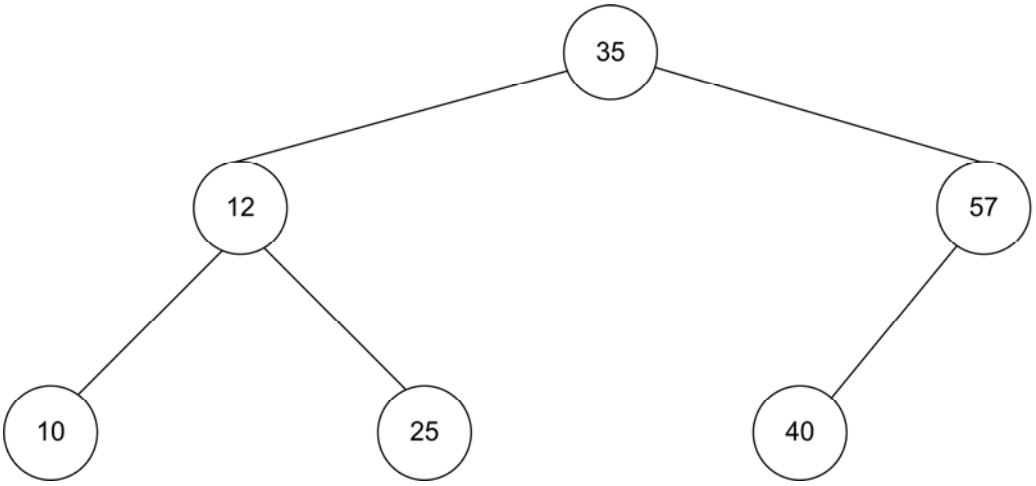
Appel	Affichage	Réponse
a) `/** Troisième affichage: Est-ce un AVL?*/`	`Ceci est un arbre AVL.` `Ceci n'est pas un arbre AVL.`	
b) `/** Cinquième affichage: Est-ce un AVL?*/`	`Ceci est un arbre AVL.` `Ceci n'est pas un arbre AVL.`	

4.7) **(5 pnt)** Considérer le code de la méthode **`isAVL()`** de l'annexe 3 et proposez une amélioration permettant d'obtenir un gain de temps d'exécution de facteur 2. Justifier votre réponse.

```
/**
 * Verifie si l'arbre est un AVL
 */
public boolean isAVL()
{
        if( !isBST(root) ) return false;
        return isAVL(root);
}

// …
private boolean isAVL(BinaryNode<AnyType> node)
{
        if (node==null) return(true);

        int h1 = height( node.left );
        int h2 = height( node.right );
        if( Math.abs( h1 - h2 ) >1 ) return false;

        if( !isAVL( node.left )) return false;
        return isAVL( node.right );
}
```

Question 5 : Arbre binaire de recherche de type AVL (18 points)

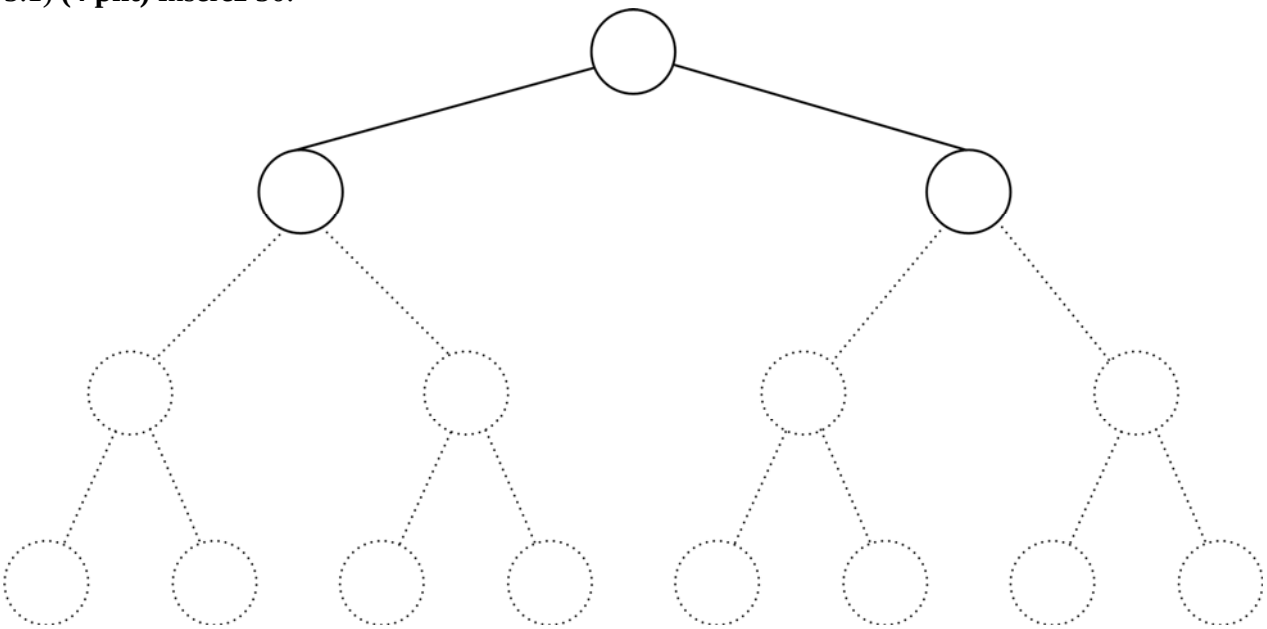
En considérant l'arbre AVL suivant :



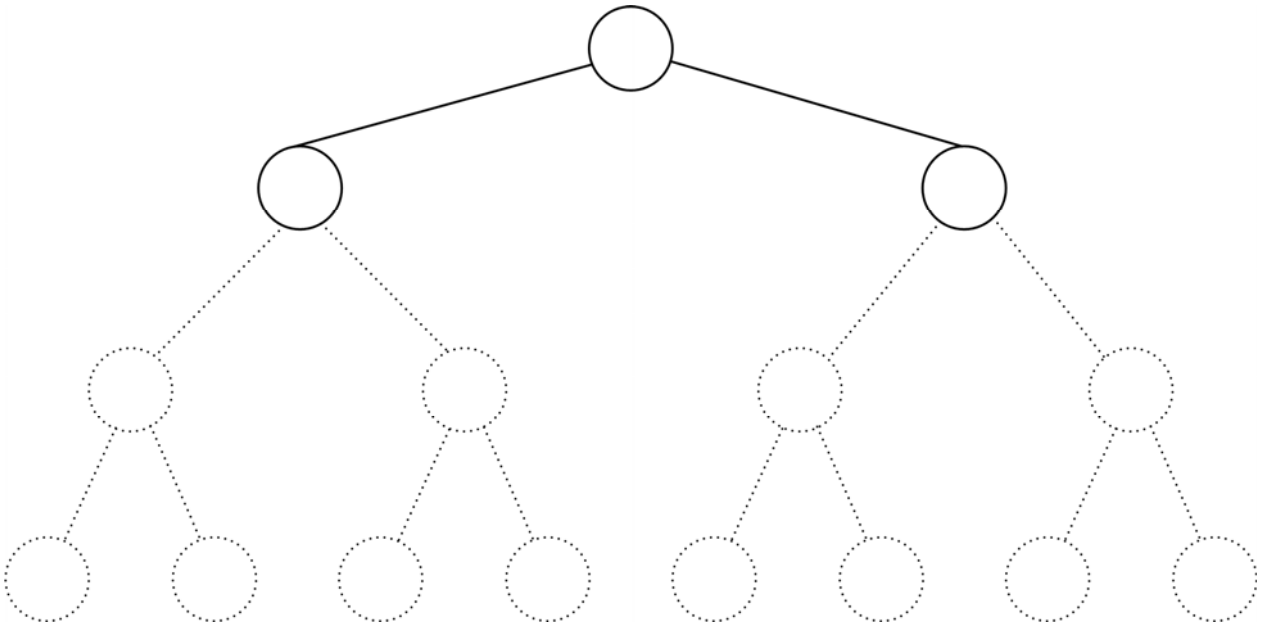
Effectuez l'ensemble des opérations suivantes dans l'ordre en vous servant des arbres ci-bas :

Insérez 50.
Insérez 37.
Insérez 45.
Insérez 52.
Insérez 47.

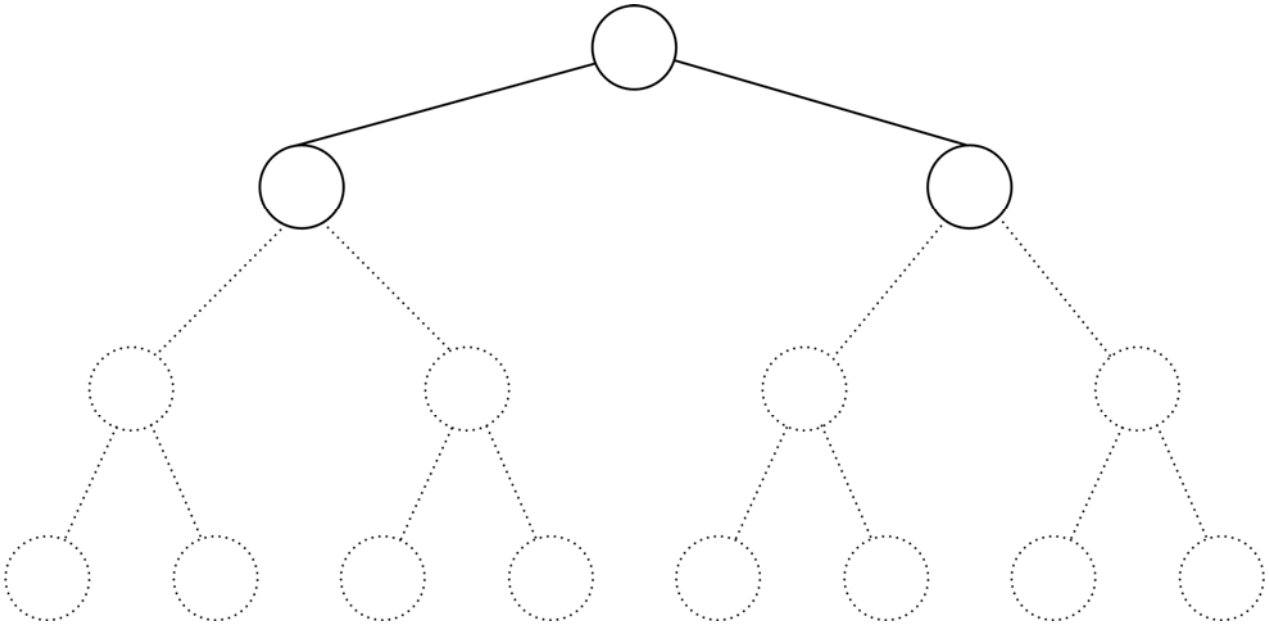
5.1) **(4 pnt)** Insérez 50.



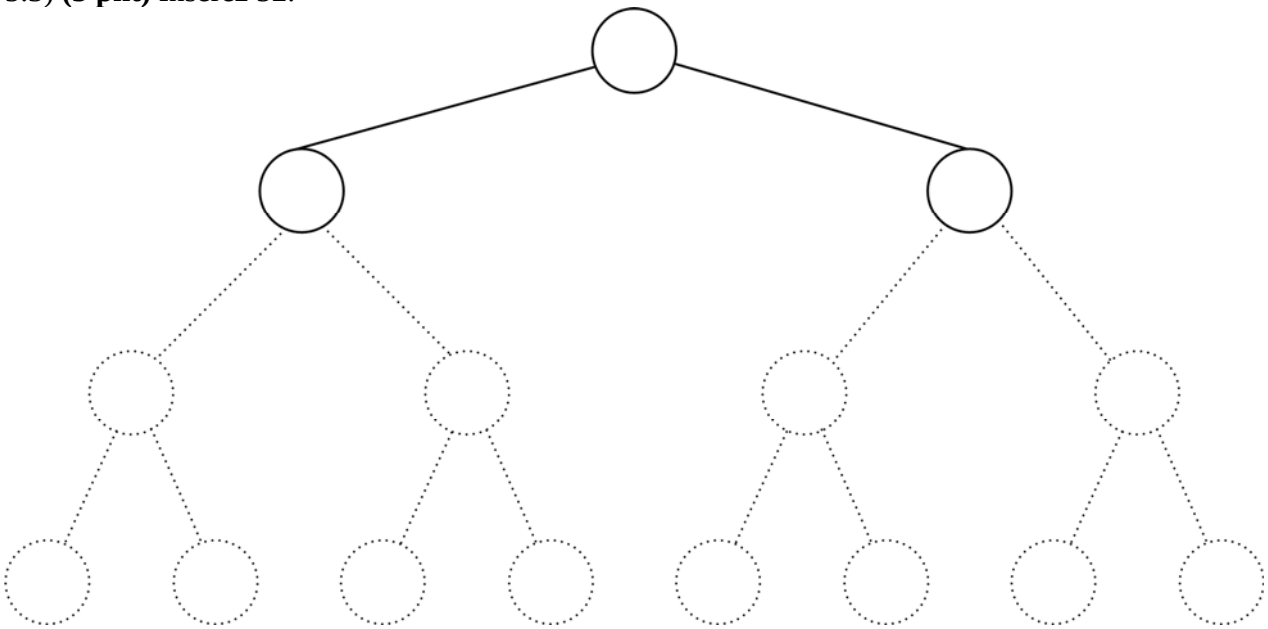
5.2) **(3 pnt)** Insérez 37.



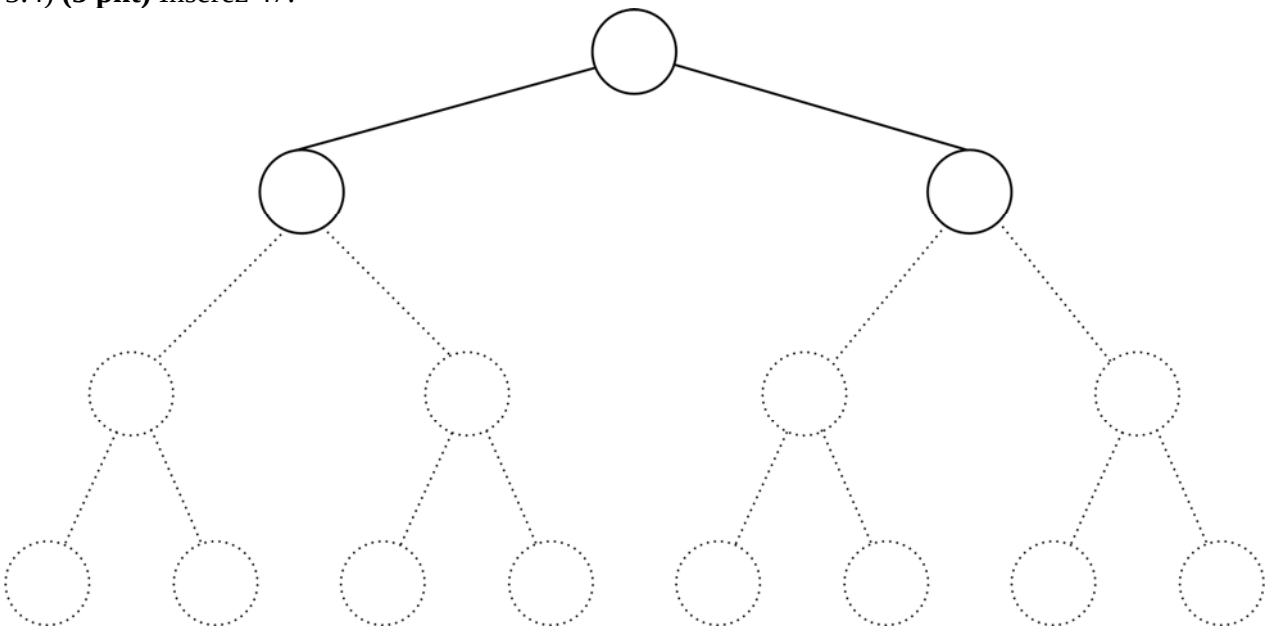
5.3) **(3 pnt)** Insérez 45.



5.3) **(3 pnt)** Insérez 52.



5.4) **(5 pnt)** Insérez 47.



Annexe 1

```
public class QuestionDoubleBubbleSort {

	public static void main(String[] args)
	{
		int a[] = new int[ 8 ];

		a[ 0 ] = 15; a[ 1 ] =  13; a[ 2 ] =  11; a[ 3 ] =  6;
		a[ 4 ] = 10; a[ 5 ] =   8; a[ 6 ] =  14; a[ 7 ] =  7;

		for(int i=0; i< a.length; ++i)
			System.out.print(a[ i ] + " ");
		System.out.println();

		//Affiche 15 13 11 6 10 8 14 7

		// Trier le vecteur
		DoubleBubbleSort( a );

		for(int i=0; i< a.length; ++i)
			System.out.print(a[ i ] + " ");

		System.out.println();

		// Affiche 6 7 8 10 11 13 14 15

	}

	public static void DoubleBubbleSort(int a[])
	{
		int j;
		int limit = a.length;
		int st = -1;

		/** "While: st<limit */
		while (st < limit)
		{
		    boolean flipped = false;
		    st++;
		    limit--;

		    /** "For 1: j= st .. limit -1 */
		    for (j = st; j < limit; j++)
		    {

		        if (a[j] > a[j + 1]) {
		                    int T = a[j];
		                    a[j] = a[j + 1];
		                    a[j + 1] = T;
		                    flipped = true;
		        }
		    }

		    if (!flipped) return;

		    /** "For 2: j= limit-1 .. st */
		    for (j = limit; --j >= st;)
		    {
		        if (a[j] > a[j + 1]) {
		                int T = a[j];
		                a[j] = a[j + 1];
		                a[j + 1] = T;
		                flipped = true;
		        }
		    }
		}
	}
}
```

Annexe 2

```
/**
 * Classe d'execution d'application
 */
public class QuestionBST {

	/**
	 *  Application
	 */
	public static void main( String [ ] args )
	{
		// Constante
		final int PROFONDEUR_MAX = 4;

		// Creation d'un arbre vide
		BinarySearchTree<Integer> t = new BinarySearchTree<Integer>( );

		/** Premier affichage: Etat de l'arbre*/
		System.out.println( "Premier Arbre:" );
		t.printTree( );


		/** Second affichage: Etat de l'arbre*/
		System.out.println( "\nSecond Arbre:" );

		// Remplissage de l'arbre avec des valeurs
		for(int i=0; i < PROFONDEUR_MAX; ++i)
			for(int j = ( 1 << (3-i) ); j < ( 1 << 4 ); j += ( 1 << (4-i) ) )
				t.insert( j );

		t.printTree( );


		/** Troisième affichage: Est-ce un AVL?*/
		if( t.isAVL() )
			System.out.println( "Ceci est un arbre AVL." );
		else
			System.out.println( "Ceci n'est pas un arbre AVL." );


		/** Quatrième affichage: Etat de l'arbre*/
		System.out.println( "\nTroisième Arbre:" );

		// Retirer certaines valeurs
		for(int i =  (1 << PROFONDEUR_MAX) - 1; i > (1 << 3); i -= 1)
			t.remove( i );

		t.printTree( );

		/** Cinquième affichage: Est-ce un AVL?*/
		if( t.isAVL() )
			System.out.println( "Ceci est un arbre AVL." );
		else
			System.out.println( "Ceci n'est pas un arbre AVL." );

		/** Bonne chance*/
	}
}
```

Annexe 3

```
/**
 * Classe BinarySearchTree
 */
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
{
        /** The tree root. */
        private BinaryNode<AnyType> root;

        /** Enumeration identifiant l'enfant. */
        enum ChildType{ leftChild, rightChild }

        /**
         * Classe interne implémentant un noeud binaire
         */
        private static class BinaryNode<AnyType>
        {
                // Constructors
                BinaryNode( AnyType theElement )
                {
                        this( theElement, null, null );
                }

                BinaryNode( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
                {
                        element  = theElement;
                        left     = lt;
                        right    = rt;
                }

                AnyType element;            // The data in the node
                BinaryNode<AnyType> left;   // Left child
                BinaryNode<AnyType> right;  // Right child
        }

        /*
         * METHODES PUBLIQUES
         */

        /**
         * Constructeur par défaut
         */
        public BinarySearchTree( ){   root = null; }

        /**
         * Insere x dans l'arbre
         */
        public void insert( AnyType x )
        {
                root = insert( x, root );
        }

        /**
         * Retire x dans l'arbre
         */
        public void remove( AnyType x )
        {
                root = remove( x, root );
        }

        /**
         * Trouve et renvoie le plus petit élément de l'arbre
         */
        public AnyType findMin( )
                {
                if( isEmpty( ) )
                        throw new java.lang.RuntimeException();
                return findMin( root ).element;
        }
```

```
/**
 * Trouve et renvoie le plus grand élément de l'arbre
 */
public AnyType findMax( )
{
        if( isEmpty( ) )
                throw new java.lang.RuntimeException();
        return findMax( root ).element;
}

/**
 * Vérifie si l'arbre contient x
 */
public boolean contains( AnyType x ){ return contains( x, root ); }

/**
 * Affiche l'arbre
 */
public void printTree( )
{
        printTree( root, "", ChildType.rightChild);
}

/**
 * Verifie si l'arbre est un AVL
 */
public boolean isAVL()
{
        if( !isBST(root) ) return false;
        return isAVL(root);
}

/**
 * Verifie si l'arbre est un Binary Search Tree (BST)
 */
public boolean isBST() { return(isBST(root)); }

/**
 * Vide l'arbre
 */
public void makeEmpty( ){ root = null; }

/**
 * Vérifie si l'arbre est vide
 */
public boolean isEmpty( ){ return root == null;}

/*
 * METHODES PRIVEES (pour fin de récursion)
 */

/**
 * Insere x dans l'arbre
 */
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
{
        if( t == null )
                return new BinaryNode<AnyType>( x, null, null );

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
                t.left = insert( x, t.left );
        else if( compareResult > 0 )
                t.right = insert( x, t.right );
        else
                ;  // Duplicate; do nothing
        return t;
}
```

```
/**
 * Retire x dans l'arbre
 */
private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
        if( t == null )
                return t;   // Item not found; do nothing

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
                t.left = remove( x, t.left );
        else if( compareResult > 0 )
                t.right = remove( x, t.right );
        else if( t.left != null && t.right != null )
        {
                t.element = findMin( t.right ).element;
                t.right = remove( t.element, t.right );
        }
        else
                t = ( t.left != null ) ? t.left : t.right;

        return t;
}

/**
 * Trouve et renvoie le plus petit élément de l'arbre
 */
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
        if( t == null )
                return null;
        else if( t.left == null )
                return t;
        return findMin( t.left );
}

/**
 * Trouve et renvoie le plus grand élément de l'arbre
 */
private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
        if( t != null )
                while( t.right != null )
                        t = t.right;

        return t;
}

/**
 * Vérifie si l'arbre contient x
 */
private boolean contains( AnyType x, BinaryNode<AnyType> t )
{
        if( t == null )
                return false;

        int compareResult = x.compareTo( t.element );

        if( compareResult < 0 )
                return contains( x, t.left );
        else if( compareResult > 0 )
                return contains( x, t.right );
        else
                return true;
}
```

```
/**
 * Affiche l'arbre
 */
private void printTree( BinaryNode<AnyType> t, String prefix, ChildType myChildType)
{
        System.out.print( prefix + "|__"); // un | et deux _

        if( t != null )
        {
                boolean isLeaf = (t.left == null && t.right == null);

                System.out.println( t.element );

                String _prefix = prefix;

                if( myChildType == ChildType.leftChild )
                        _prefix +=  "| "; // un | et un espace
                else
                        _prefix +=  "  " ; // deux espaces

                if( !isLeaf )
                {
                        printTree( t.left,  _prefix, ChildType.leftChild );
                        printTree( t.right, _prefix, ChildType.rightChild );
                }
        }
        else
                System.out.print("null\n");
}

/**
 * Verifie si l'arbre est un AVL
 */
private boolean isAVL(BinaryNode<AnyType> node)
{
        if (node==null) return(true);

        int h1 = height( node.left );
        int h2 = height( node.right );
        if( Math.abs( h1 - h2 ) >1 ) return false;

        if( !isAVL( node.left )) return false;
        return isAVL( node.right );
}

/**
 * Verifie si l'arbre est un Binary Search Tree (BST)
 */
private boolean isBST(BinaryNode<AnyType> node)
{
        if (node==null) return(true);

        if ( node.left!=null )
                if( findMax(node.left).element.compareTo(node.element) > 0 )
                        return false;

        if ( node.right!=null )
                if( findMin(node.right).element.compareTo(node.element) < 0 )
                        return false;

        if( !isBST(node.left) ) return false;

        return(  isBST(node.right) );

}
```

```
        /**
         * Donne la hauteur d'un (sous-)arbre de racine t
         */
        private int height( BinaryNode<AnyType> t )
        {
                if( t == null )
                        return 0;
                else
                        return 1 + Math.max( height( t.left ), height( t.right ) );
        }
}
```