POLYTECHNIQUE MONTRÉAL

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**	**20173**
Professeur		***Local***	***Téléphone***
Ettore Merlo et Tarek Ould Bachir			
Jour	***Date***	***Durée***	***Heure***
Lundi	**16 octobre 2017**	**2h00**	**18h00**

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 **6** questions sur un total de **23** 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 (18 points)

Soit une table de dispersement avec sondage quadratique Hash(clé) = (clé + i^2) % N dont l'implémentation est fournie à l'Annexe 1 à titre de référence. Sachant que les cinq (5) clés suivantes ont été insérées dans cet ordre :

81, 29, 56, 68, 69.

1.1) **(5 points)** Trouvez les deux clés qui ne sont pas à leur place dans la table de taille N=13 présentée ci-après :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Entrées				81	29				69		68		56

Clés mal positionnées : _____,_____

1.2) **(2 points)** Replacez à leur bonne position les clés identifiées à la question 1.1) et donnez ci-après l'état de la table après correction:

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

1.3) **(2 points)** Quelle a été la plus grande valeur prise par *i* (le *i* de Hash(clé) = (clé + i^2) % N) lors de l'insertion des cinq clés précédentes ? Vous pouvez vous aider du code source fourni à l'Annexe 1.

Plus grande valeur prise par *i* : _____

1.4) **(2 points)** Après l'insertion des cinq clés précédentes, on effectue un appel à `remove(29)`. Donnez le détail de cet appel. Soyez bref mais précis. Vous pouvez vous aider du code source fourni à l'Annexe 1.

1.5) **(1 points)** Quelle sera la plus grande valeur prise par *i* (le *i* de Hash(clé) = (clé + i^2) % N) lors de l'appel `remove(29)`.

Plus grande valeur prise par *i* : _____

1.6) **(2 points)** Après l'appel à `remove(29)` à la question 1.4), on effectue un appel à `remove(81)`. Donnez le détail de cet appel. Soyez bref mais précis. Vous pouvez vous aider du code source fourni à l'Annexe 1.

1.7) **(2 points)** Donnez l'état de la table après l'appel à de `remove(81)` de la question 1.6):

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

1.8) **(2 points)** Après l'appel à `remove(81)` à la question 1.6), on effectue un appel à `insert(42)`. Donnez l'état de la table après cette insertion. Vous pouvez vous aider du code source fourni à l'Annexe 1.

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

Question 2 : Tri fusion (25 points)

Partie I : On désire exécuter l'algorithme `mergeSort` pour trier le vecteur ci-après. Le code source vous est fourni à l'Annexe 2.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Valeurs	3	2	4	15	12	10	7	13	19	21	12	17	12	5	3

2.1) (**5 points**) Donnez, dans l'ordre d'appel, l'ensemble des dix (10) premières valeurs que prennent les paramètres `left` et `right` lors des appels successifs à la méthode privée `mergeSort`. Aidez-vous du code de l'Annexe 2.

Pour précision, il est bien question de la méthode dont la signature est :

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

Appels	**`left`**	**`right`**
Appel 1		
Appel 2		
Appel 3		
Appel 4		
Appel 5		
Appel 6		
Appel 7		
Appel 8		
Appel 9		
Appel 10		

2.2) (**4 points**) À la fin de l'exécution de l'algorithme, quelle aura été la seconde plus grande taille de vecteur sur laquelle la méthode privée `mergeSort` aura été appelée, la plus grande étant 15. Aidez-vous du code de l'Annexe 2.

Pour précision, il est bien question de la méthode dont la signature est :

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

Taille du second plus grand vecteur sur lequel `mergeSort` est appelé : _____

2.3) (**4 points**) Positionnez dans le tableau fourni ci-après les éléments du vecteur identifié à la question 2.2) entre les positions `left` et `right` (inclusivement) au moment d'entrer dans la méthode privée `mergeSort`.

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

2.4) (**4 points**) Positionnez dans le tableau fourni ci-après les éléments du vecteur identifié à la question 2.2) entre les positions `left` et `right` (inclusivement) au moment de sortir de la méthode privée `mergeSort`.

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

Partie II : On vous propose une variation sur l'implémentation de l'algorithme `mergeSort` où la méthode privée est donnée comme suit :

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

      if( left < right ){

         int center = reverse(a, left, right);

         mergeSort( a, tmpArray, left, center );
         mergeSort( a, tmpArray, center + 1, right );

         merge( a, tmpArray, left, center + 1, right );
      }
   }
```

où :

```
private static <AnyType extends Comparable<? super AnyType>>
int reverse( AnyType[] a, int left, int right ){
   int center = ( left + right ) / 2;

   for(int i=left, j=right; i<=center; i++, j--)
      swapReferences( a, i, j );

   return center;
}

private static <AnyType> void swapReferences(AnyType[] a, int idx1, int idx2 ){
   AnyType tmp = a[ idx1 ];
   a[ idx1 ]   = a[ idx2 ];
   a[ idx2 ]   = tmp;
}
```

2.5) (**2 points**) Quelle est la complexité en pire cas de cette nouvelle implémentation ? Justifiez brièvement.

2.6) (**3 points**) Positionnez dans le tableau fourni ci-après les éléments du vecteur `a` au moment de sortir du premier appel à la méthode `reverse(…)`.

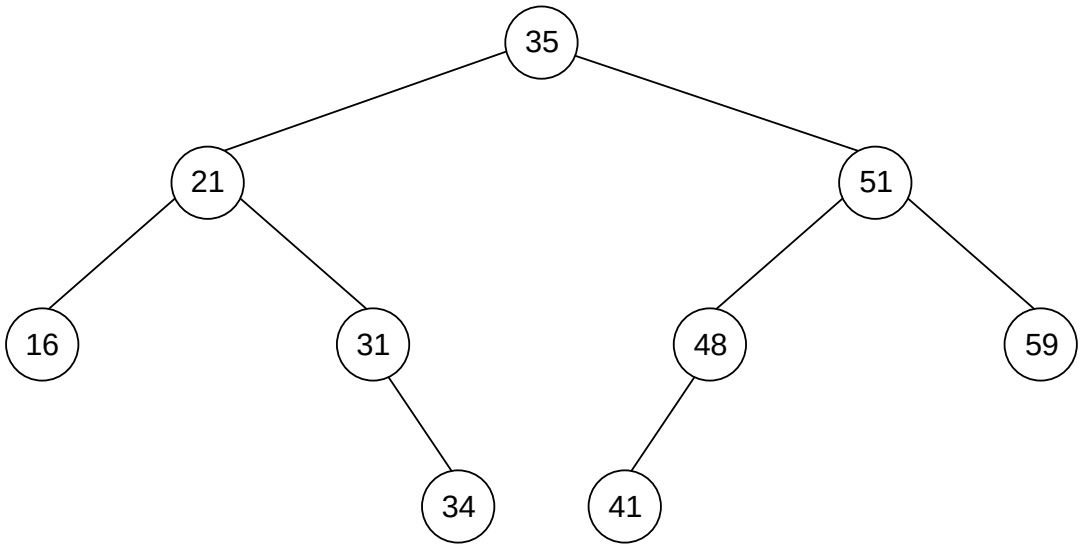
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Valeurs															

2.7) (**3 points**) Positionnez dans le tableau fourni ci-après les éléments du vecteur `a` au moment de sortir du second appel à la méthode `reverse(…)`.

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

Question 3 : Parcours d'arbres (12 points)

Considérez le code de l'Annexe 3 ainsi que l'arbre binaire suivant. La méthode `toString()` est appelée sur cet arbre.



3.1) (**4 points**) Donnez la valeur du booléen `startWithLeft` passé en paramètre à la méthode `printTree(…)` en même temps que chacun des nœuds de l'arbre lors de l'appel à la méthode `toString()`. Par exemple, la racine (nœud 35) est passée en paramètre avec la valeur `true`.

Répondez en remplissant le tableau suivant.

Nœud	`startWithLeft`
35	**`true`**
21	
51	
16	
31	
48	
59	
34	
41	

3.2) (**2.5 points**) Si la variable `BinaryNode<Integer> t` contient une référence vers le nœud 21 de l'arbre précédent, donnez le contenu de la chaîne de caractères générée à la fin de l'exécution du code suivant :

```
StringBuilder s = new StringBuilder();
printTree(t, s, false);
s.toString();
```

Réponse:

3.3) (**2.5 points**) Si la variable `BinaryNode<Integer> t` contient une référence vers le nœud 51 de l'arbre précédent, donnez le contenu de la chaîne de caractères générée à la fin de l'exécution du code suivant :

```
StringBuilder s = new StringBuilder();
printTree(t, s, false);
s.toString();
```

Réponse:

3.4) (**3 points**) Donnez le résultat de l'affichage de l'arbre binaire précédent lors de l'appel à la méthode `toString()`. Référez-vous au code de l'Annexe 3.

Réponse:

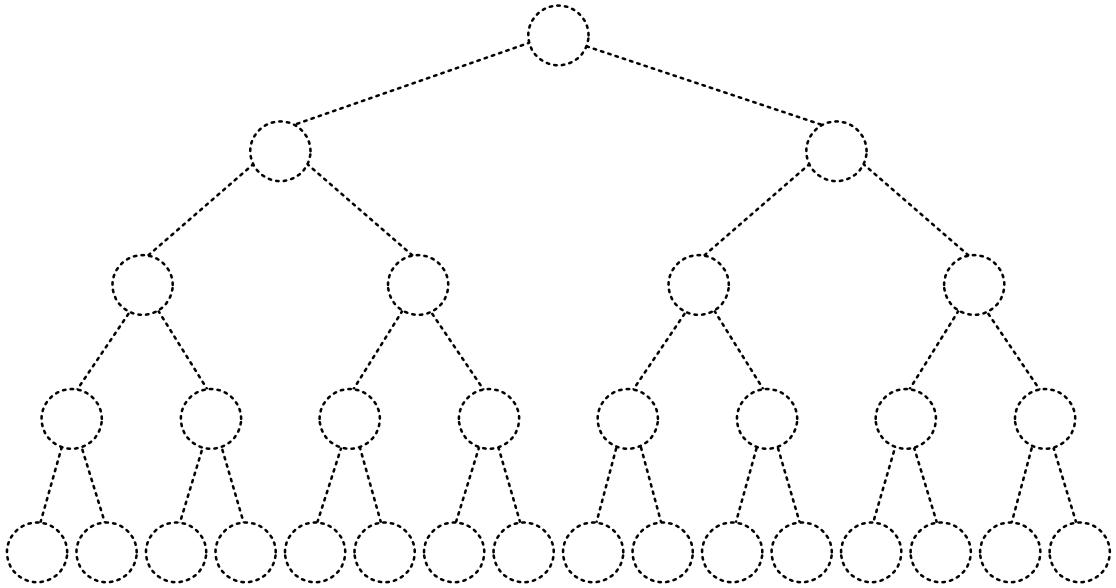
Question 4 : Arbres binaire de recherche (13 points)

Note : Tel que vu en classe, l'ordre de traversement des enfants d'un nœud se fait de gauche à droite selon la figure.

4.1) **(3 points)** Si l'affichage par niveaux d'un arbre binaire de recherche donne :

51, 26, 43, 35, 47, 37, 45, 50

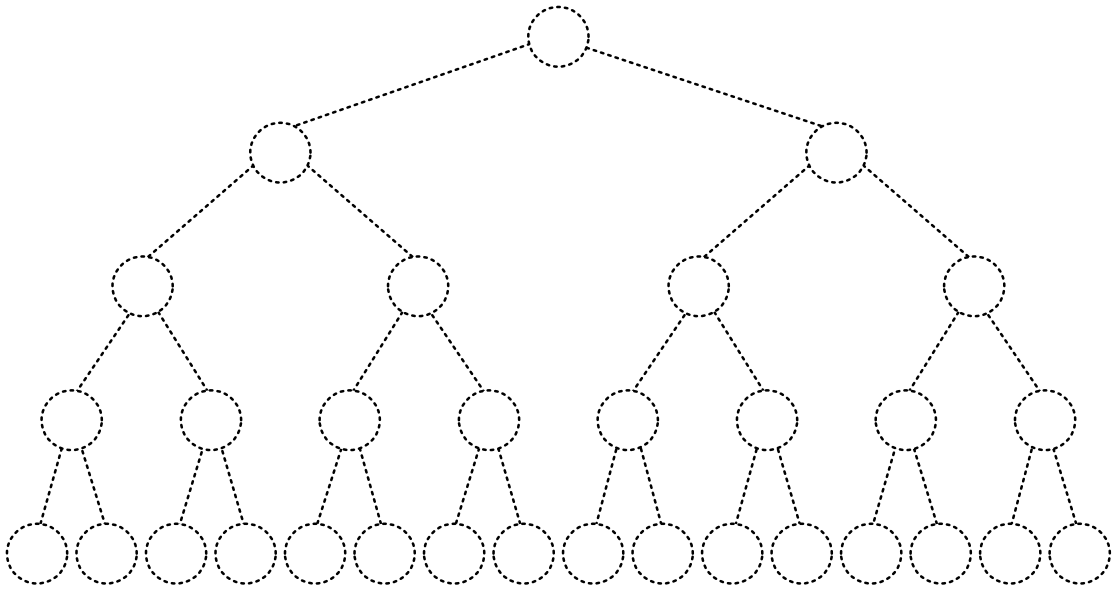
Donnez sa représentation graphique :



4.2) **(3 points)** Si l'affichage pré-ordre d'un arbre binaire de recherche donne :

65, 31, 86, 71, 78, 92, 89

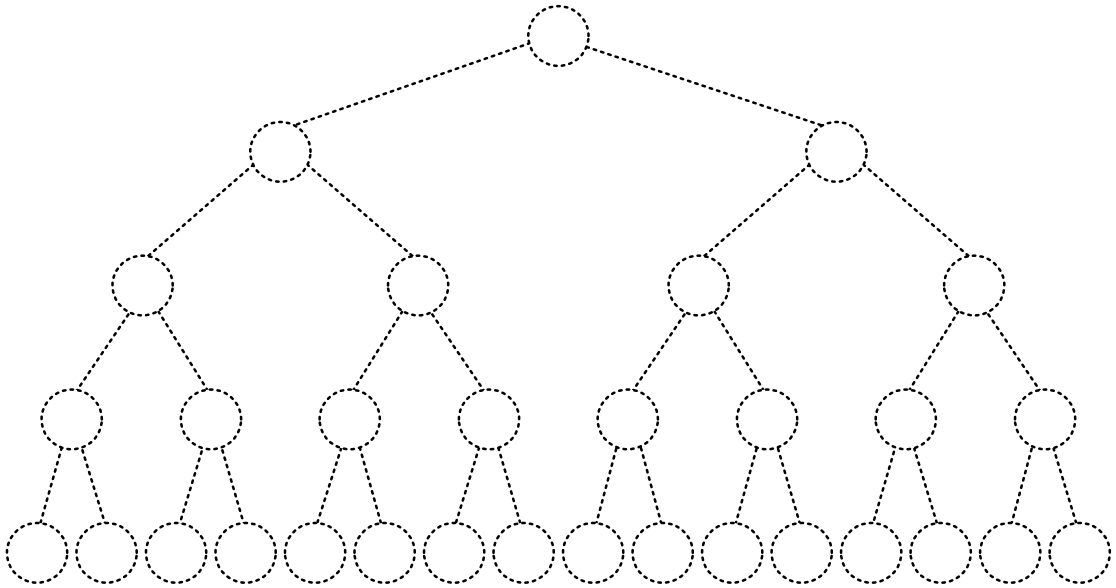
Donnez sa représentation graphique :



4.3) **(3 points)** Si l'affichage post-ordre d'un arbre binaire de recherche donne :

42, 13, 67, 86, 84, 91, 95, 87, 56

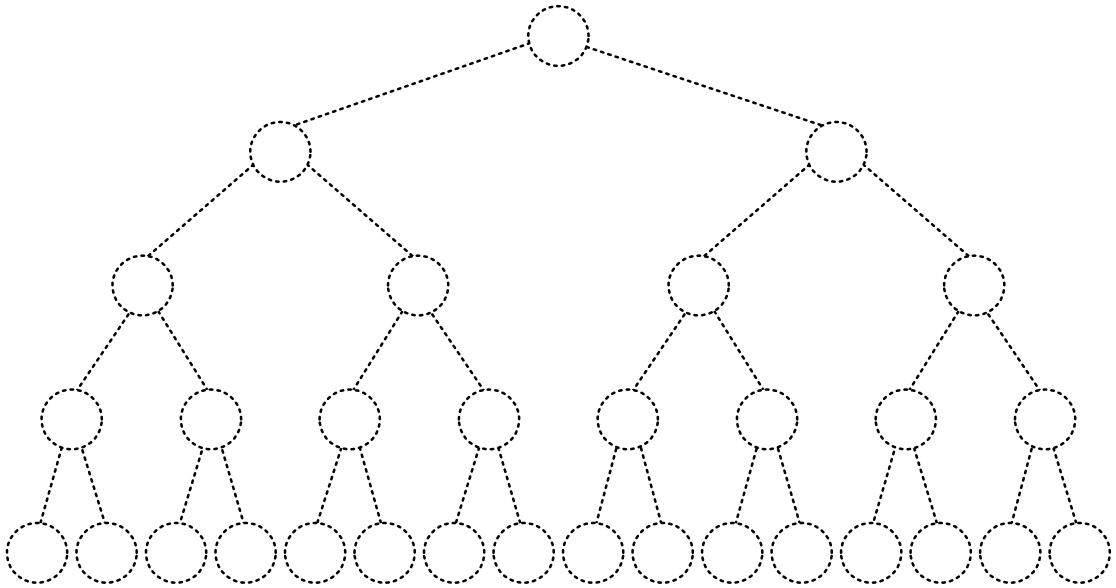
Donnez sa représentation graphique :



4.4) **(4 points)** Si l'affichage en-ordre d'un arbre binaire de recherche donne :

7, 10, 21, 32, 43, 54, 65, 76, 87, 98, 99

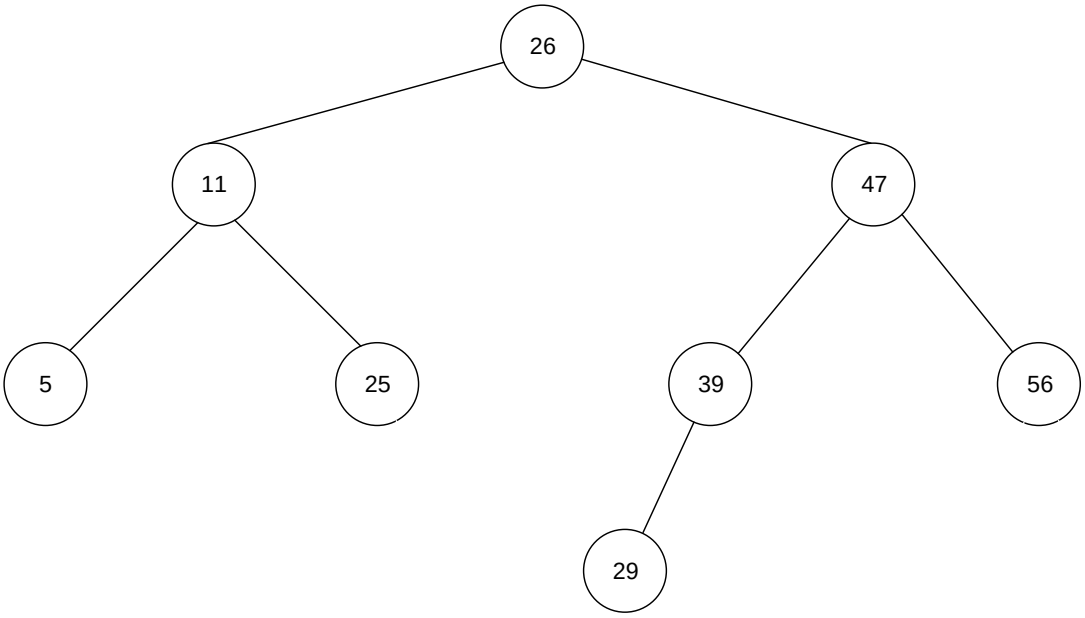
Donnez sa représentation graphique sachant qu'il s'agit d'un AVL dont la racine est 76.



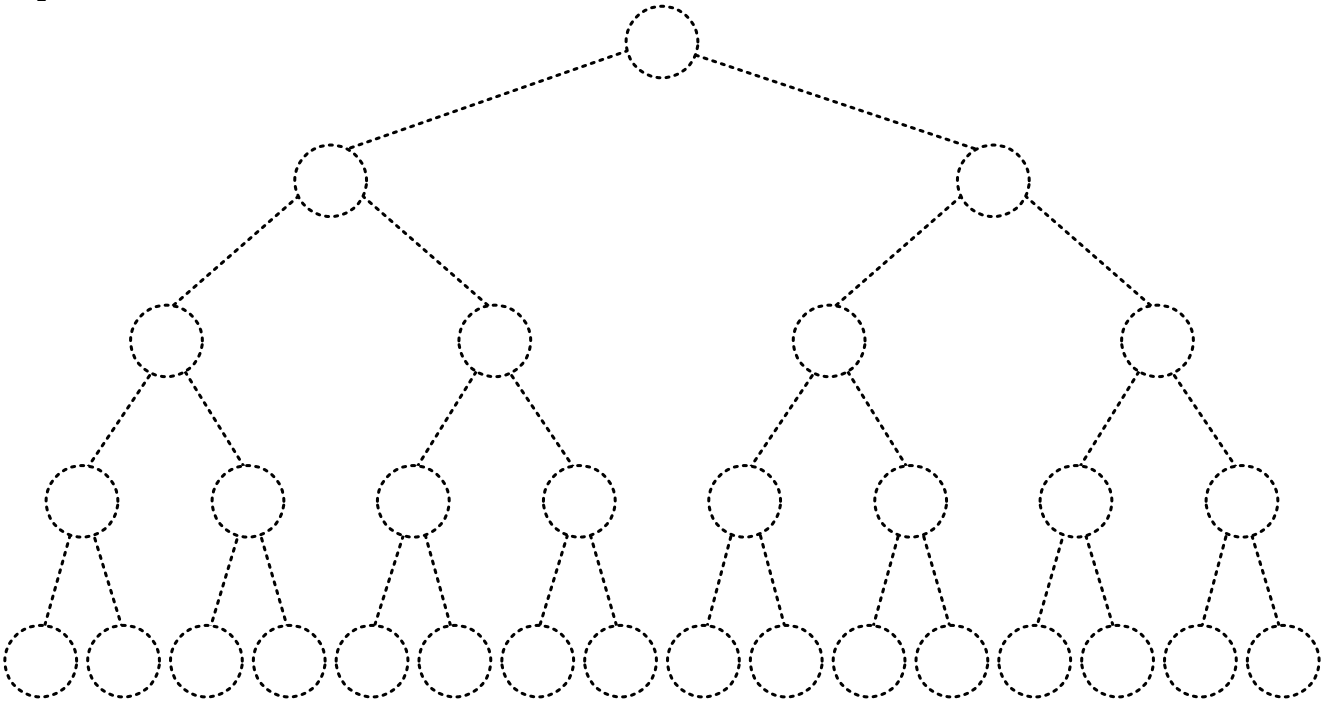
Question 5 : Arbre binaire de recherche de type AVL (19 points)

En partant de chacun des arbres AVL suivants, effectuer les opérations demandées.

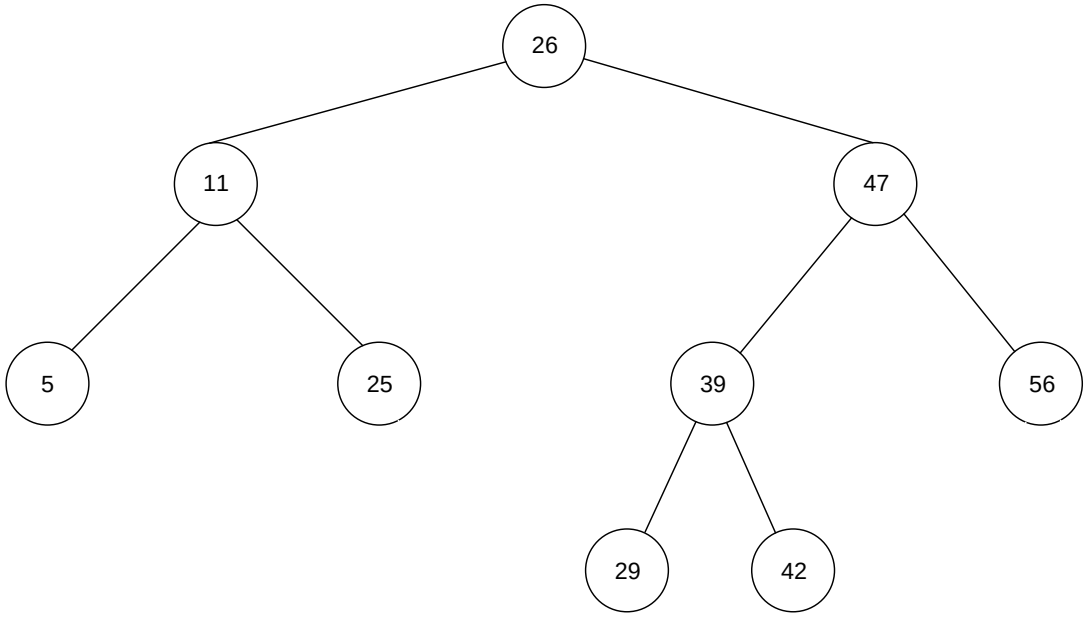
5.1) **(2 points)** Insérez 27.



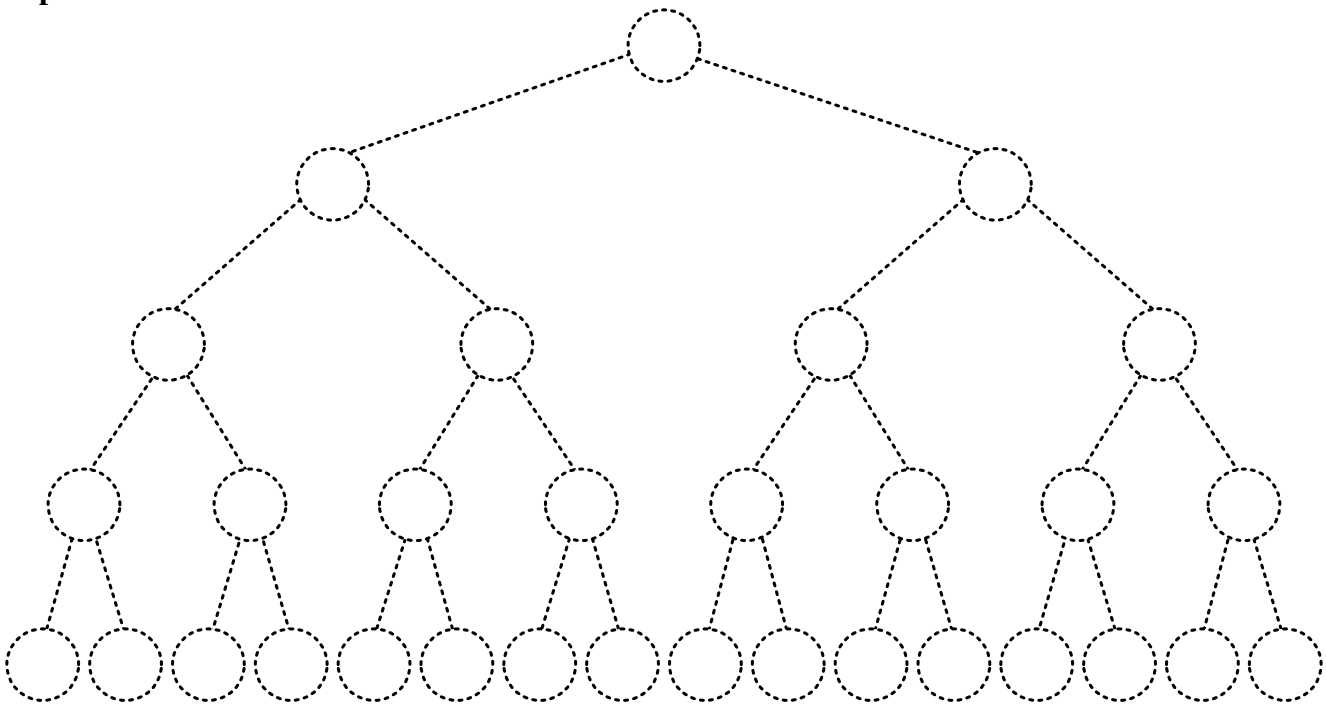
Réponse :



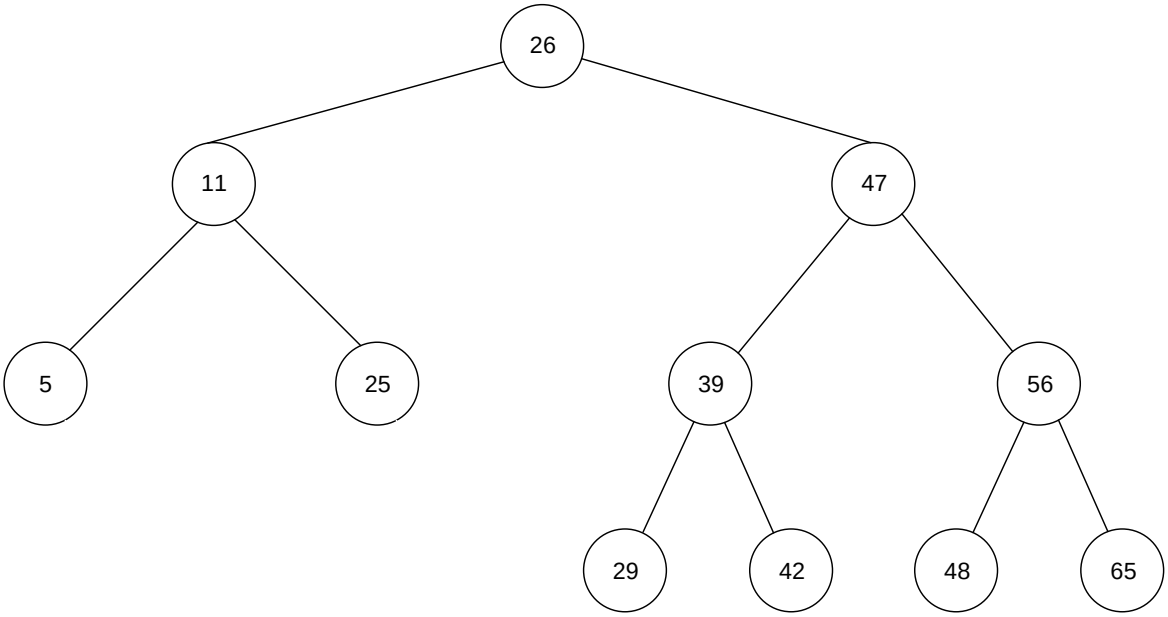
5.2) **(3 points)** Insérez 45.



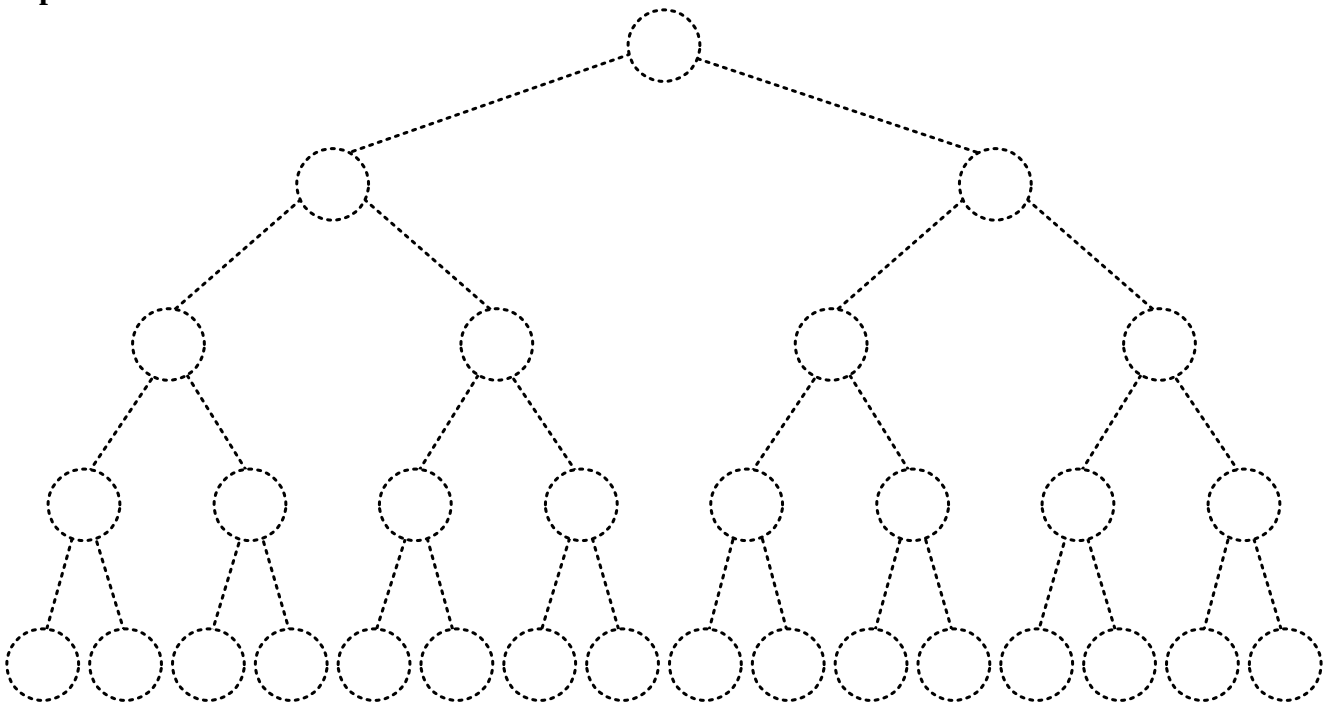
Réponse :



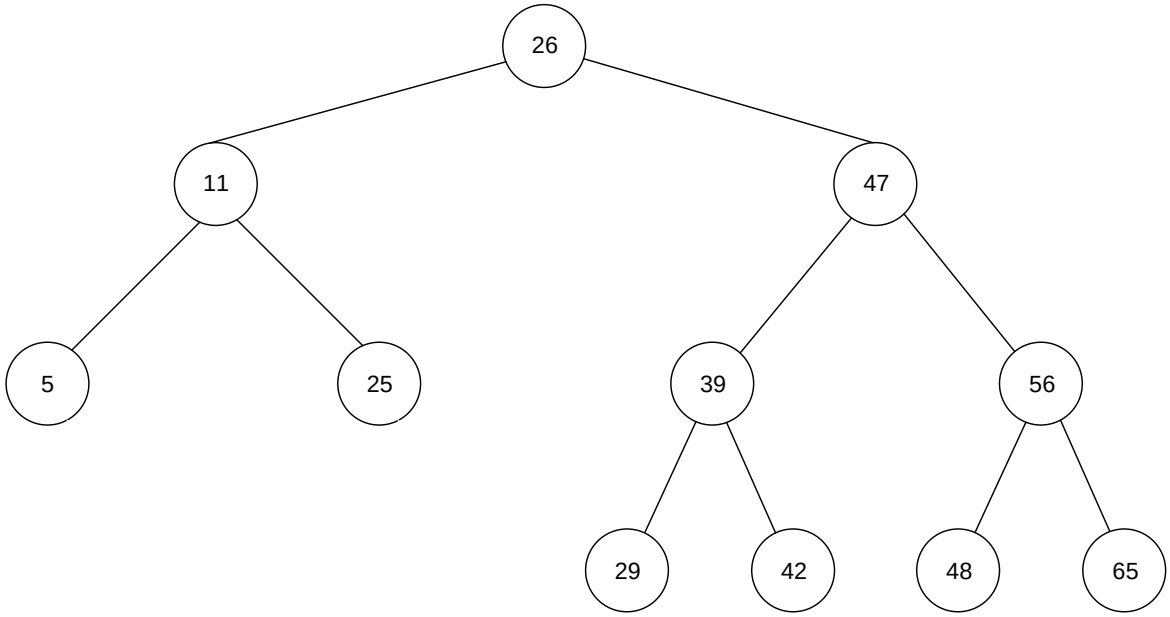
5.3) **(3 points)** Insérez 72.



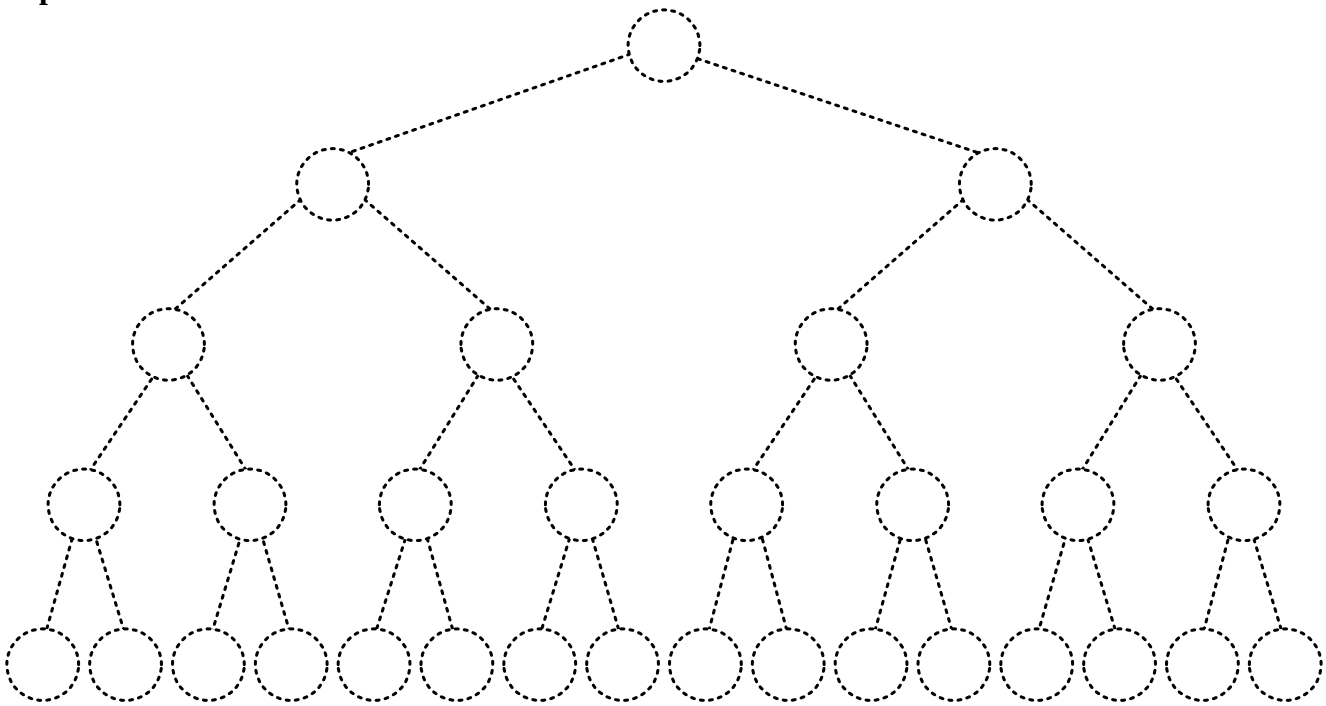
Réponse :



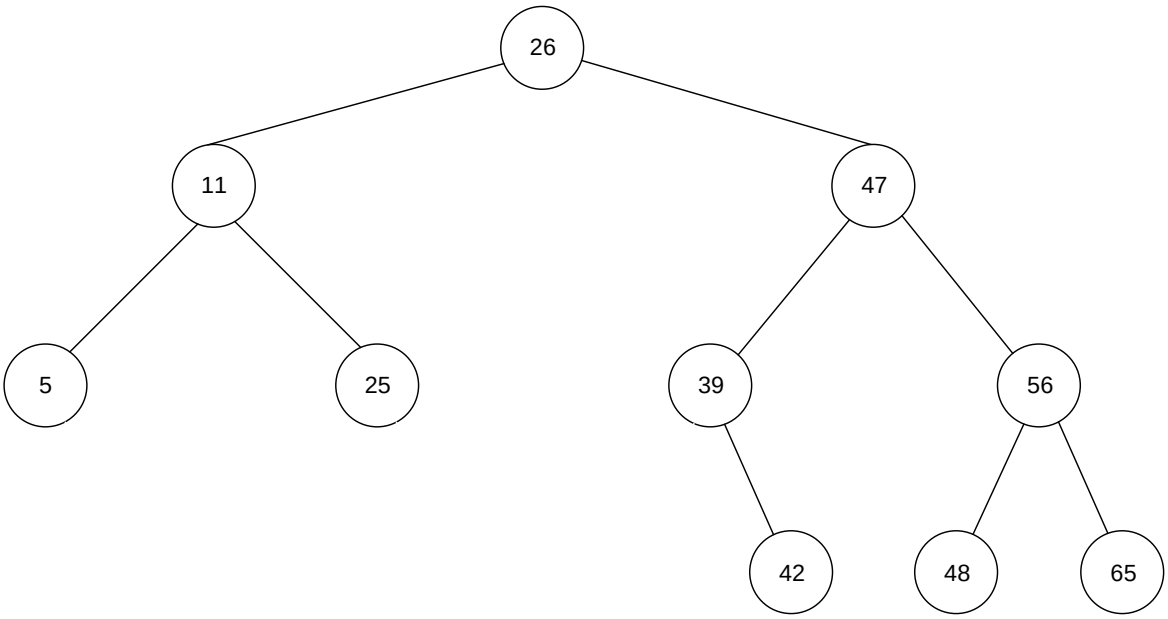
5.4) **(3 points)** Insérez 28.



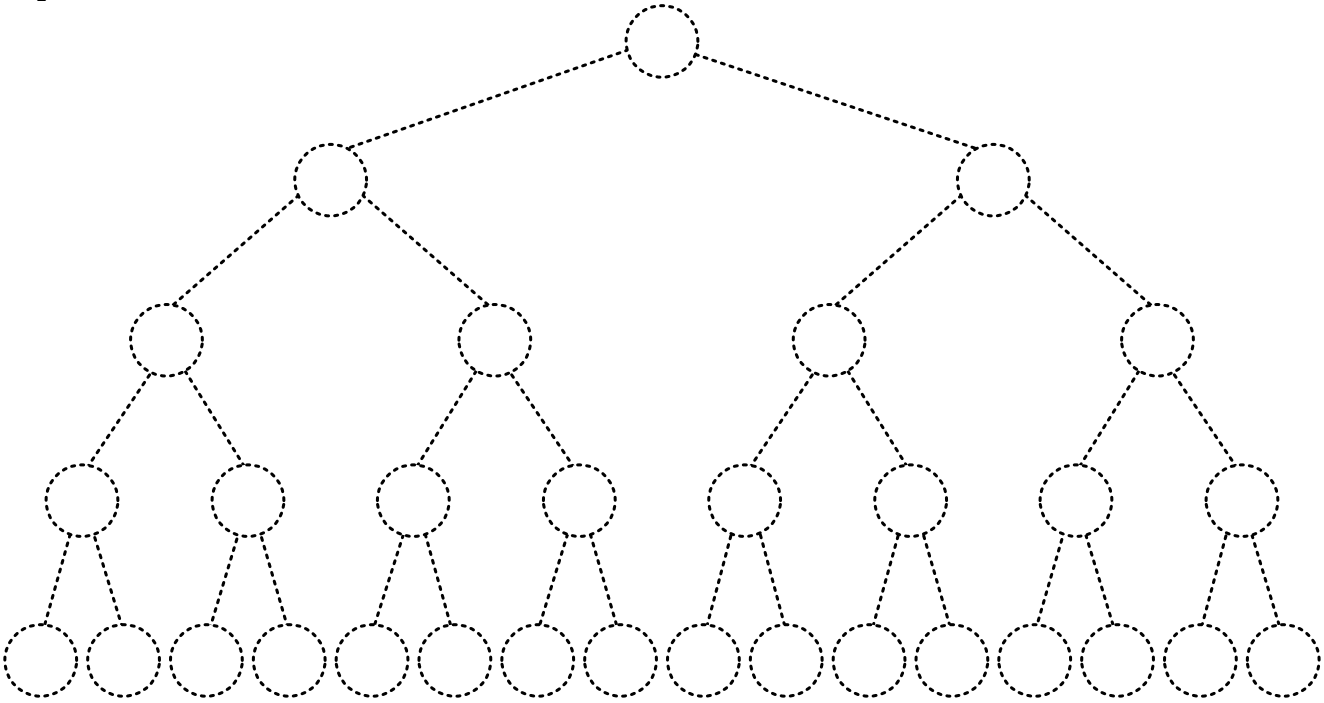
Réponse :



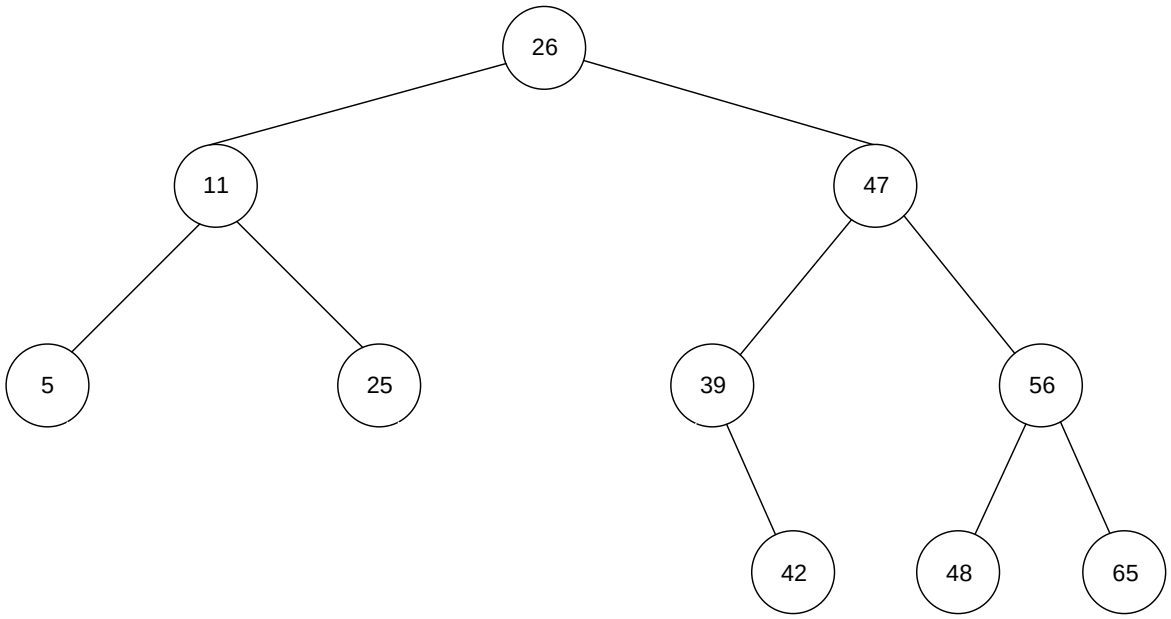
5.5) **(2.5 points)** Retirez la racine (utilisez la méthode `remove` d'un arbre binaire de recherche standard telle que vue en classe).



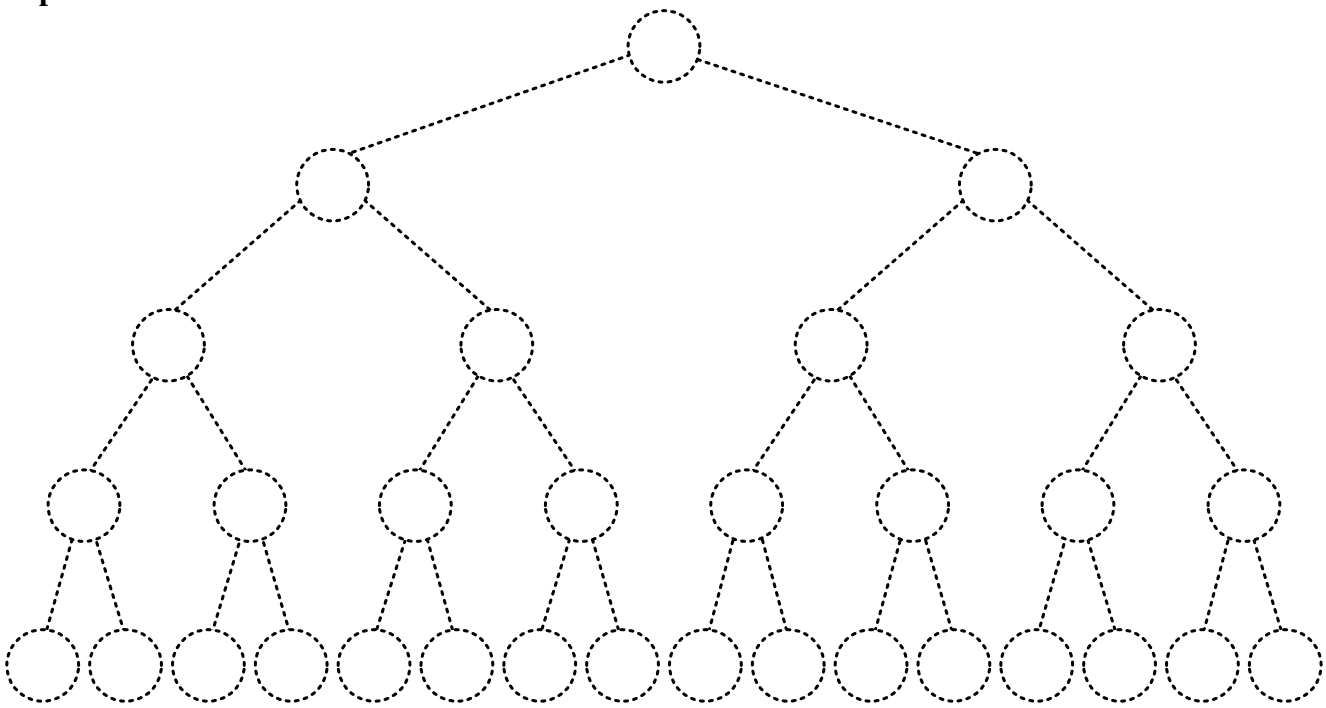
Réponse :



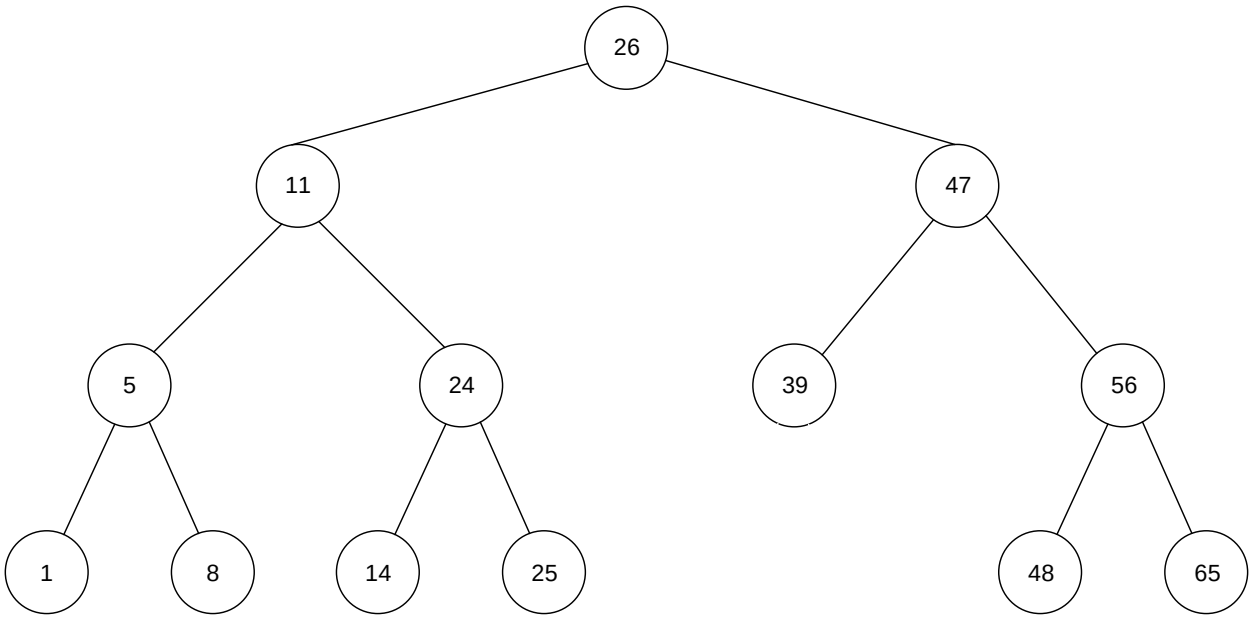
5.6) **(2.5 points)** Retirez le nœud 47 (utilisez la méthode `remove` d'un arbre binaire de recherche standard telle que vue en classe).



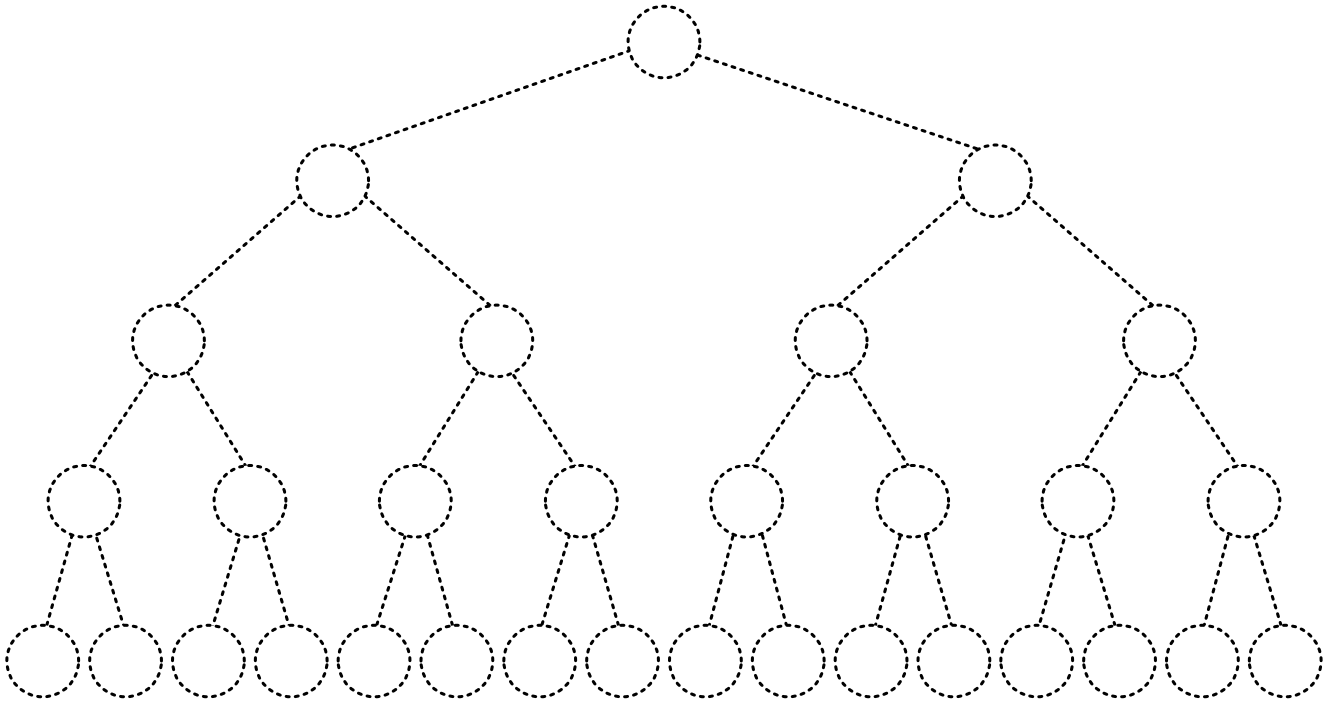
Réponse :



5.7) **(3 points)** Effectuer une rotation simple à un des nœuds de l'arbre pour obtenir un arbre complet.



Réponse :
Nœud choisi :_____, Direction de la rotation :_____
Résultat :



Question 6 : Généralités (13 points)

Répondez aux assertions suivantes par « vrai » ou par « faux ». Justifier votre réponse brièvement. Les réponses non justifiées ne seront pas considérées.

6.1) **(2 points)** La fusion de deux listes triées en ordre inverse en une liste unique triée en ordre peut s'effectuer en $O(n)$.

6.2) **(2.5 points)** Si la liste `liste` est une `ArrayList`, le code suivant aura une complexité $O(n^2)$ en pire cas.

```
public static void dupliquerCles (List<Integer> liste, int cle)
{
   int i = 0;
   while( i < liste.size() ){
      if( liste.get( i ).hashCode() == cle )
         liste.add( i++, cle );
      i++;
   }
}
```

6.3) **(2.5 points)** Si la liste `list` est une `ArrayList`, le code suivant aura une complexité $O(n^2)$ en pire cas.

```
public static void dupliquerCles (List<Integer> liste, int cle )
{
   ListIterator<Integer> it = liste.listIterator();
   while( it.hasNext() )
      if( it.next().hashCode() == cle )
         it.add(cle);
}
```

6.4) **(2 points)** En meilleur cas, l'algorithme `mergeSort` a une complexité O($n \log(n)$).

6.5) **(2 points)** Un arbre binaire de recherche de type Splay a nécessairement une hauteur proportionnelle à log(n), où n est le nombre de nœuds dans l'arbre.

6.6) **(2 points)** Un arbre AVL de hauteur h=7 possède au moins 55 nœuds.

Annexe 1

```
public class QuadraticProbingHashTable<AnyType>{

   public QuadraticProbingHashTable(){ this( DEFAULT_TABLE_SIZE ); }

   public QuadraticProbingHashTable( int size ){
      allocateArray( size );
      makeEmpty( );
   }

   public void insert( AnyType x ){
      int currentPos = findPos( x );
      if( isActive( currentPos ) ) return;
      array[ currentPos ] = new HashEntry<AnyType>( x, true );

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

   private void rehash( ){
      HashEntry<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 );
   }

   private int findPos( AnyType x ){
      int offset = 1;
      int currentPos = myhash( x );

      while( array[ currentPos ] != null &&
      !array[ currentPos ].element.equals( x ) ){
         currentPos += offset;
         offset += 2;
         if( currentPos >= array.length )
         currentPos -= array.length;
      }

      return currentPos;
   }

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

   public boolean contains( AnyType x ){
      return isActive(findPos( x ));
   }
```

```
   private boolean isActive( int currentPos ){
      return array[ currentPos ] != null && array[ currentPos ].isActive;
   }

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

   private int myhash( AnyType x ){
      int hashVal = x.hashCode( );

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

      return hashVal;
   }

   private static class HashEntry<AnyType>{
      public AnyType  element;
      public boolean isActive;

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

   private static final int DEFAULT_TABLE_SIZE = 13;
   private HashEntry<AnyType> [ ] array;
   private int currentSize;

   @SuppressWarnings("unchecked")
   private void allocateArray( int arraySize ){
      array = new HashEntry[ nextPrime( arraySize ) ];
   }

   private static int nextPrime( int n ){
      if( n <= 0 ) n = 3;
      if( n % 2 == 0 ) n++;
      for( ; !isPrime( n ); n += 2 ) ;
      return n;
   }

   private static boolean isPrime( int n ){
      if( n==1 || n == 2 || n == 3 || n % 2 == 0  ) return true;

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

      return true;
   }
}
```

Annexe 2

```
public final class Sort{

   @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 );
   }

   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 );
      }
   }

   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;

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

      while( leftPos <= leftEnd ){ tmpArray[ tmpPos++ ] = a[ leftPos++ ]; }

      while( rightPos <= rightEnd ){ tmpArray[ tmpPos++ ] = a[ rightPos++ ]; }

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

Annexe 3

```
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>{

   public BinarySearchTree( ){ root = null; }

   public void insert( AnyType x ){ root = insert( x, root ); }

   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;
   }

   public String toString( ){
      StringBuilder s = new StringBuilder();
      if( root == null )
      s.append( "Empty tree" );
      else
      printTree( root, s, true);

      return s.toString();
   }

   private void printTree( BinaryNode<AnyType> t, StringBuilder s,
                           boolean startWithLeft ){
      if( t != null ) {
         if( startWithLeft ){
            printTree( t.left, s, true);
            s.append(t.element + ", ");
            printTree( t.right, s, false);
         }
         else{
            printTree( t.right, s, true);
            s.append( t.element + ", ");
            printTree( t.left, s, false);
         }
      }
   }
```

```
   private static class BinaryNode<AnyType>{
      BinaryNode(AnyType theElement){ this( theElement, null, null ); }

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

      AnyType element;            // Donnée dans le noeud
      BinaryNode<AnyType> left;   // Enfant de gauche
      BinaryNode<AnyType> right;  // Enfant de droite
   }

   private BinaryNode<AnyType> root;

   public static void main( String [ ] args ){
      BinarySearchTree<Integer> t = new BinarySearchTree<Integer>( );
      t.insert( 35 );
      t.insert( 21 );
      t.insert( 51 );
      t.insert( 16 );
      t.insert( 31 );
      t.insert( 48 );
      t.insert( 59 );
      t.insert( 34 );
      t.insert( 41 );

      System.out.println( t );
   }
}
```