POLYTECHNIQUE MONTRÉAL

Corrigé 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**	**20163**
Professeur		***Local***	***Téléphone***
Ettore Merlo et Tarek Ould Bachir			
Jour	***Date***	***Durée***	***Heure***
Lundi	**17 octobre 2016**	**2h00**	**18h30**

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 **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 (17 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 six (6) clés suivantes ont été insérées dans cet ordre :

54, 32, 41, 16, 67, 84.

1.1) **(5 points)** Trouvez la clé qui n'est pas à sa 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			54	41	16		32	84					67

Clé mal positionnée : 67.

1.2) **(2 points)** Replacez à sa bonne position la clé identifiée 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			54	41	16		32	84				67	

1.3) **(3 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 six clés précédentes ? Vous pouvez vous aider du code source fourni à l'Annexe 1.

Plus grande valeur prise par *i* : 3.

1.4) **(4 points)** Après l'insertion des six clés précédentes, on effectue un appel à `remove(28)`. Donnez le détail de cet appel. Soyez bref mais précis. Vous pouvez vous aider du code source fourni à l'Annexe 1.

28%13 = 2, collision;
(2+ 1)%13 = 3, collision;
(3+3)% 13 = 6, collision;
(6+5)%13 = 11, collision;
(11+7)%13 = 5, case est vide. 28 est absent. On sort.

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(28)`.

Plus grande valeur prise par *i* :4

1.6) **(2 point)** Combien de clés supplémentaires faut-il ajouter à la table de dispersement pour que la fonction `rehash()` soit appelée?

Une clé.

Question 2 : Tri fusion (30 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**
Valeurs	13	2	4	5	12	6	7	13	9	11	12	7	2

2.1) (**10 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	0	12
Appel 2	0	6
Appel 3	0	3
Appel 4	0	1
Appel 5	0	0
Appel 6	1	1
Appel 7	2	3
Appel 8	2	2
Appel 9	3	3
Appel 10	4	6

2.2) (**4 points**) À la fin de l'exécution de l'algorithme, quelle aura été la plus grande taille de vecteur sur laquelle la méthode privée `mergeSort` aura été appelée. 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 plus grand vecteur sur lequel `mergeSort` est appelé : 13

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**
Valeurs	13	2	4	5	12	6	7	13	9	11	12	7	2

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**
Valeurs	2	2	4	5	6	7	7	9	11	12	12	13	13

Partie II : On vous propose une variation sur l'implémentation de l'algorithme `mergeSort` où la méthode privé 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 = median3( 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 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 );
   return center
}

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

2.5) (**4 points**) Quelle est la complexité en pire cas de cette nouvelle implémentation ? Justifiez brièvement.

La fonction median3 a un temps d'exécution constant par itération, de sorte que la complexité en pire cas demeure O(nlg(n)).

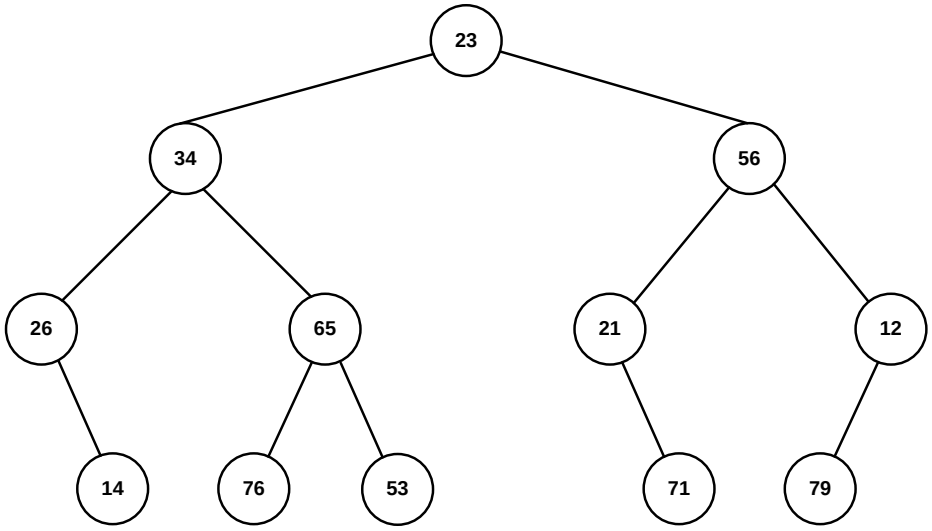
2.6) (**4 points**) Quelle est la complexité en meilleur cas de cette nouvelle implémentation ? Justifiez brièvement.

La fonction median3 a un temps d'exécution constant par itération, de sorte que la complexité en meilleur cas demeure O(nlg(n)).

Question 3 : Parcours d'arbres (10 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.

Considérez l'arbre binaire suivant:



3.1) (**2.5 points**) Donner le résultat de l'affichage de l'arbre binaire si il est parcouru en pré-ordre. Séparez les éléments par des virgules.

23, 34, 26, 14, 65, 76, 53, 56, 21, 71, 12, 79

3.2) (**2.5 points**) Donner le résultat de l'affichage de l'arbre binaire si il est parcouru en post-ordre. Séparez les éléments par des virgules.

14, 26, 76, 53, 65, 34, 71, 21, 79, 12, 56, 23

3.3) (**2.5 points**) Donner le résultat de l'affichage de l'arbre binaire si il est parcouru en ordre. Séparez les éléments par des virgules.

26, 14, 34, 76, 65, 53, 23, 21, 71, 56, 79, 12

3.4) (**2.5 points**) Donner le résultat de l'affichage de l'arbre binaire si il est parcouru par niveaux. Séparez les éléments par des virgules.

23, 34, 56, 26, 65, 21, 12, 14, 76, 53, 71, 79

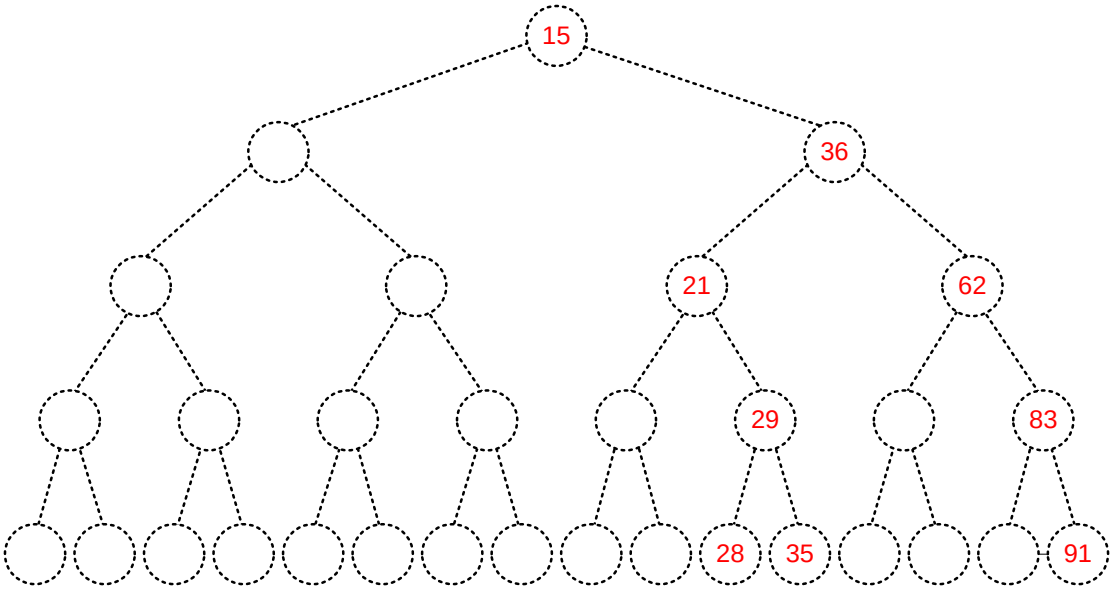
Question 4 : Arbres binaire de recherche (18 points)

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

15, 36, 21, 62, 29, 83, 28, 35, 91

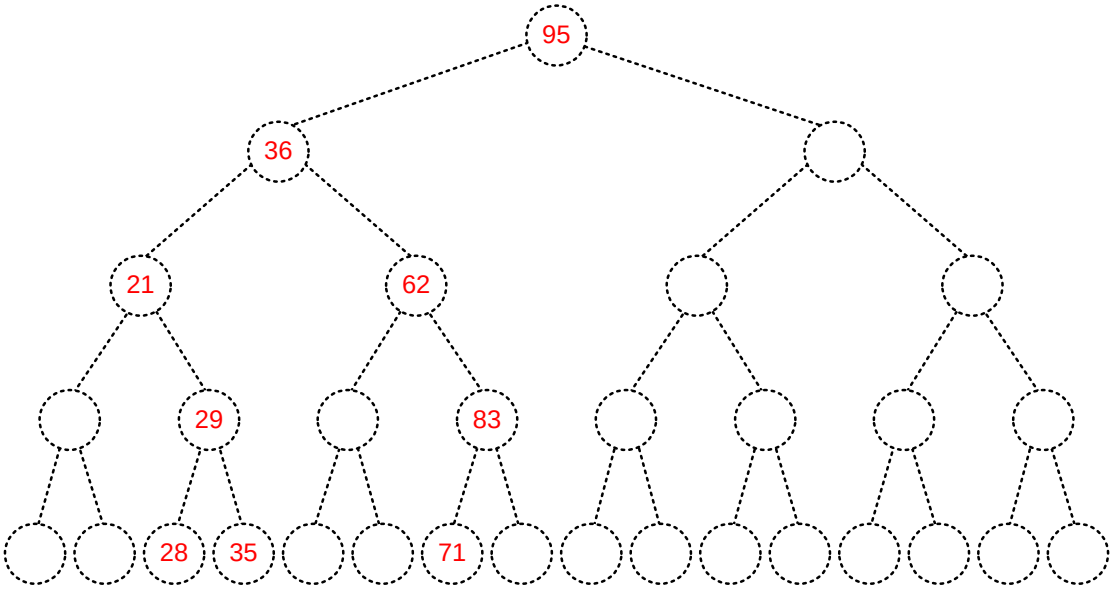
Donnez sa représentation graphique :



4.2) **(4 points)** Si l'affichage pré-ordre d'un arbre binaire de recherche donne :

95, 36, 21, 29, 28, 35, 62, 83, 71

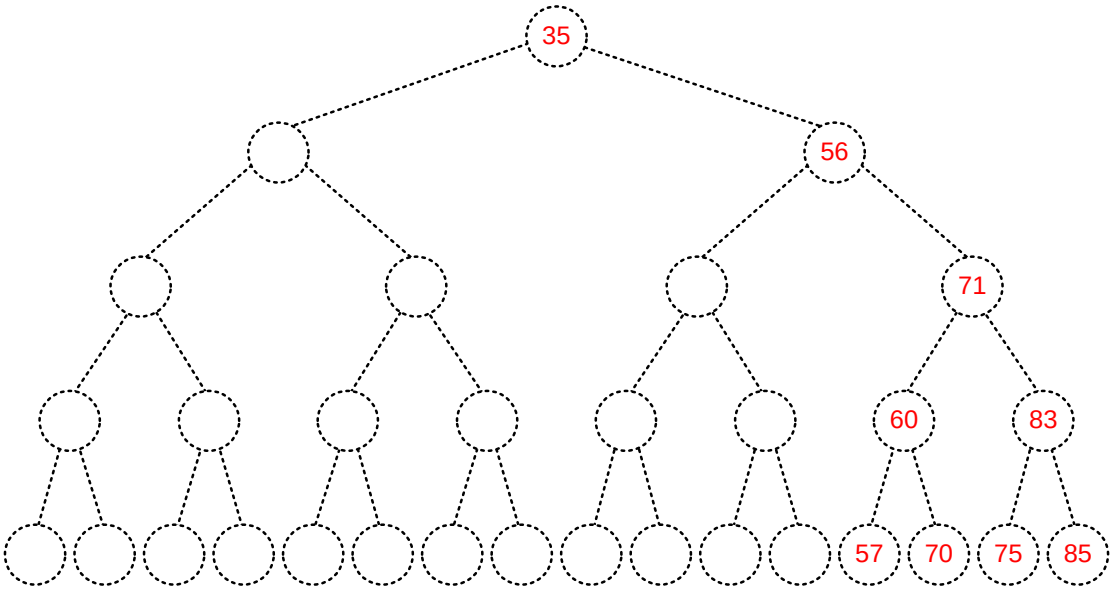
Donnez sa représentation graphique :



4.3) **(4 points)** Si l'affichage post-ordre d'un arbre binaire de recherche donne :

57, 70, 60, 75, 85, 83, 71, 56, 35

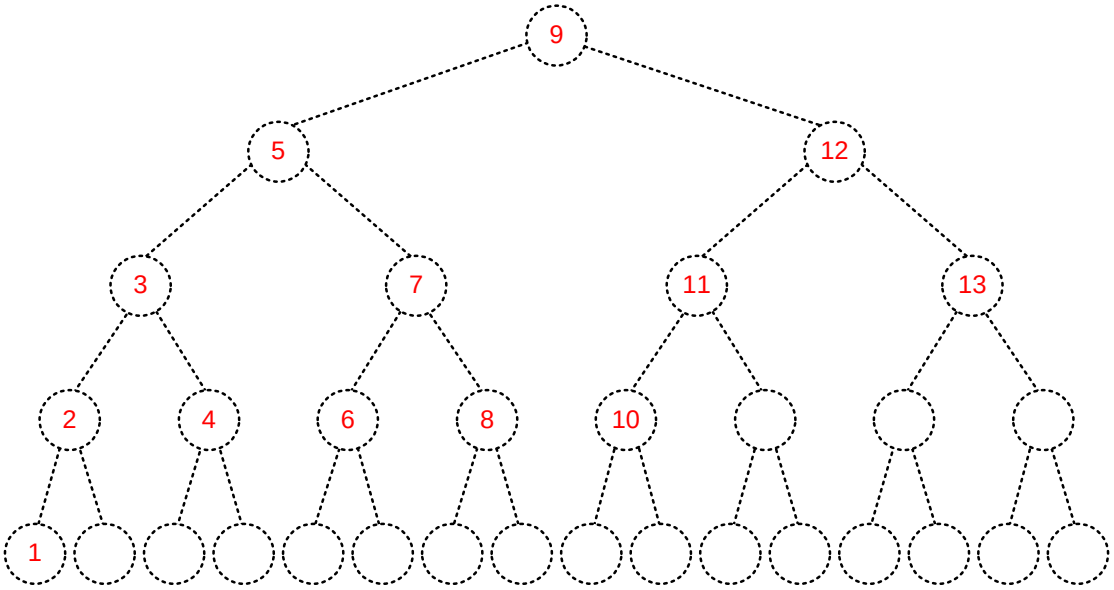
Donnez sa représentation graphique :



4.4) **(6 points)** Si l'affichage en-ordre d'un arbre binaire de recherche donne :

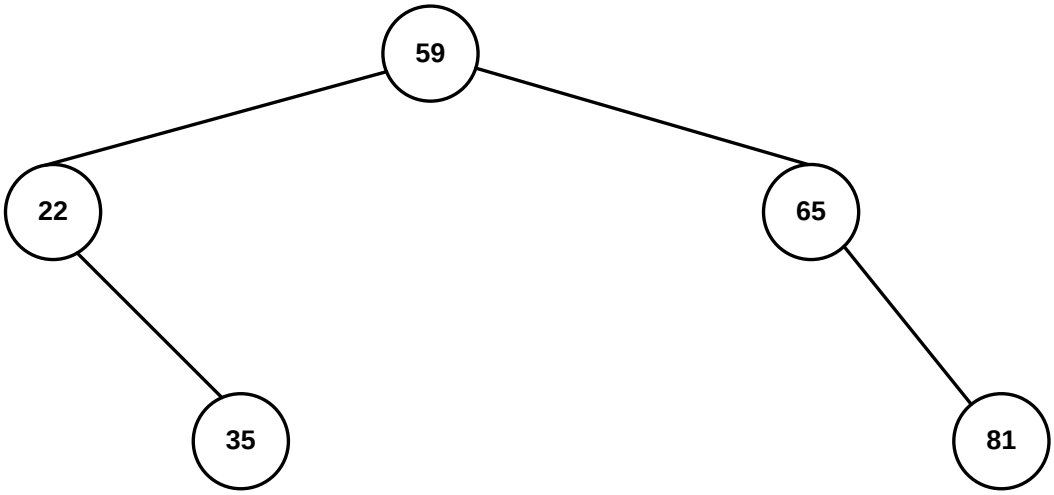
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

Donnez sa représentation graphique sachant que les sous-arbres à la gauche et à la droite de la racine sont des arbres complets, que le sous-arbre de gauche a une hauteur de 3 et que le sous-arbre de droite a une hauteur de 2.



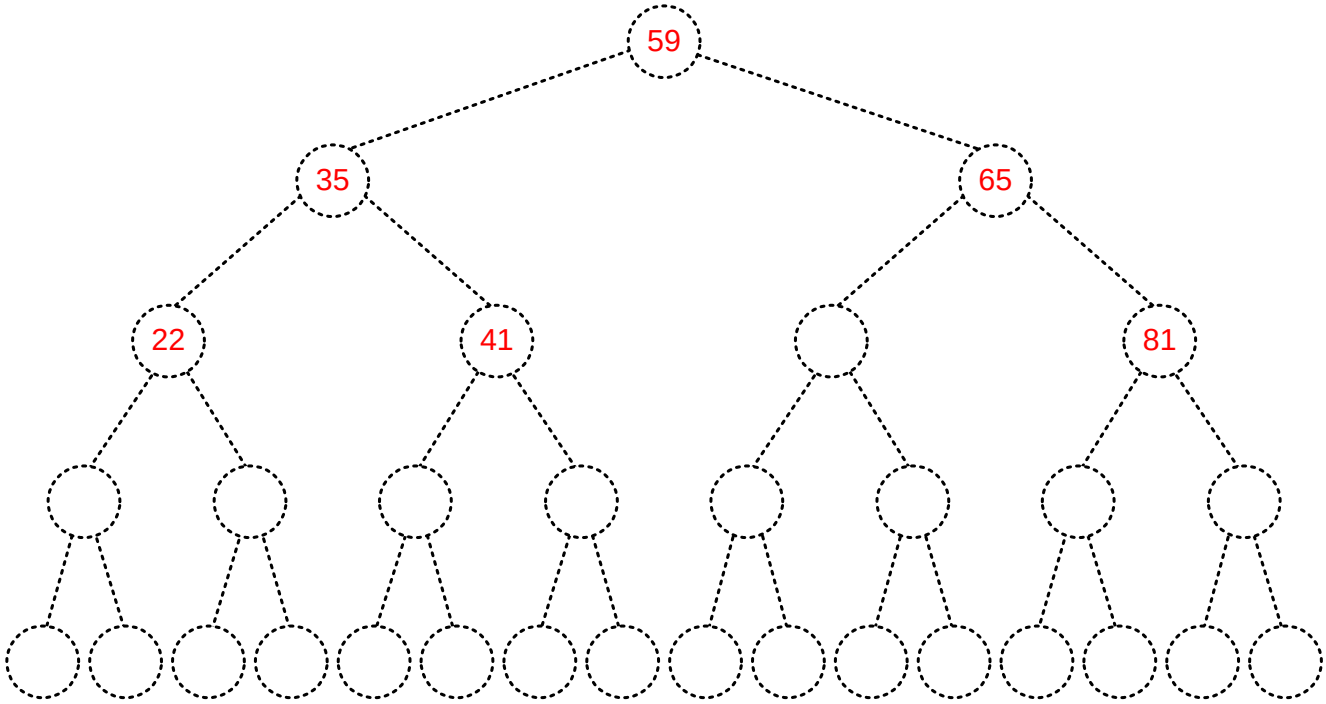
Question 5 : Arbre binaire de recherche de type AVL (15 points)

En partant de l'arbre AVL suivant :

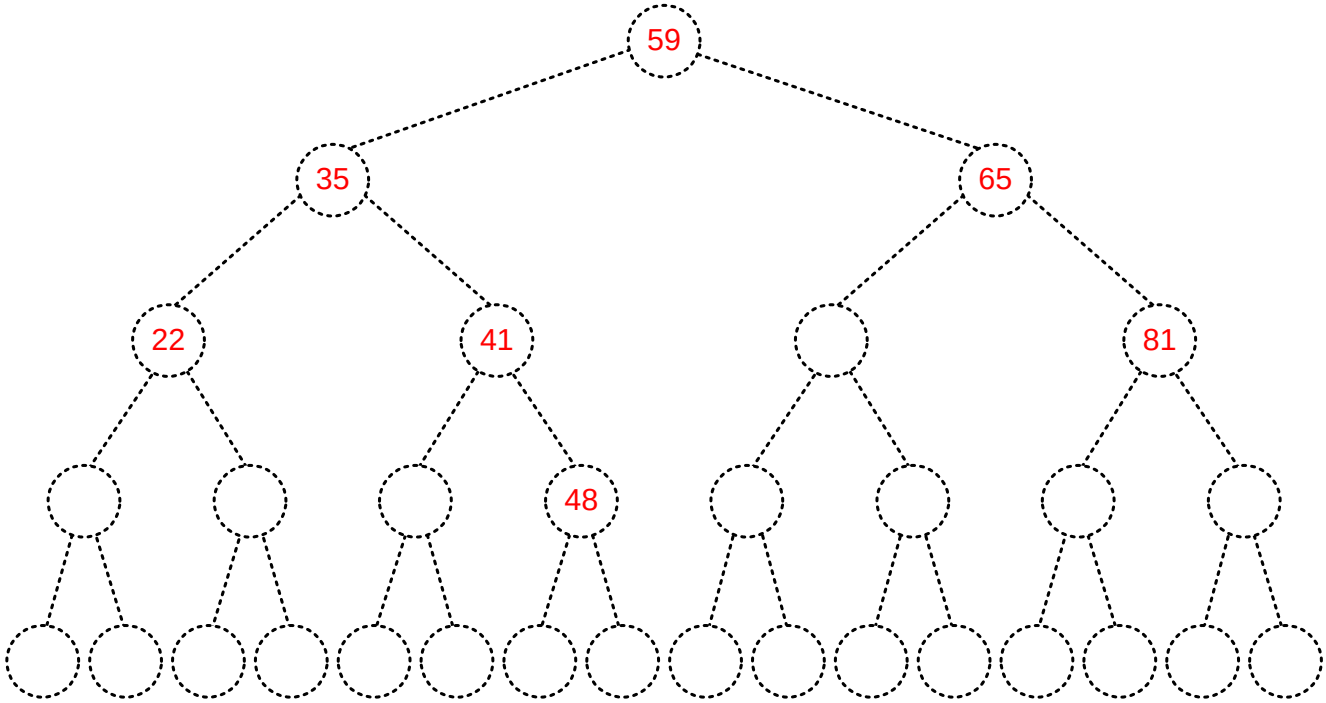


Insérez dans l'ordre les clés suivantes : 41, 48, 54, 85.

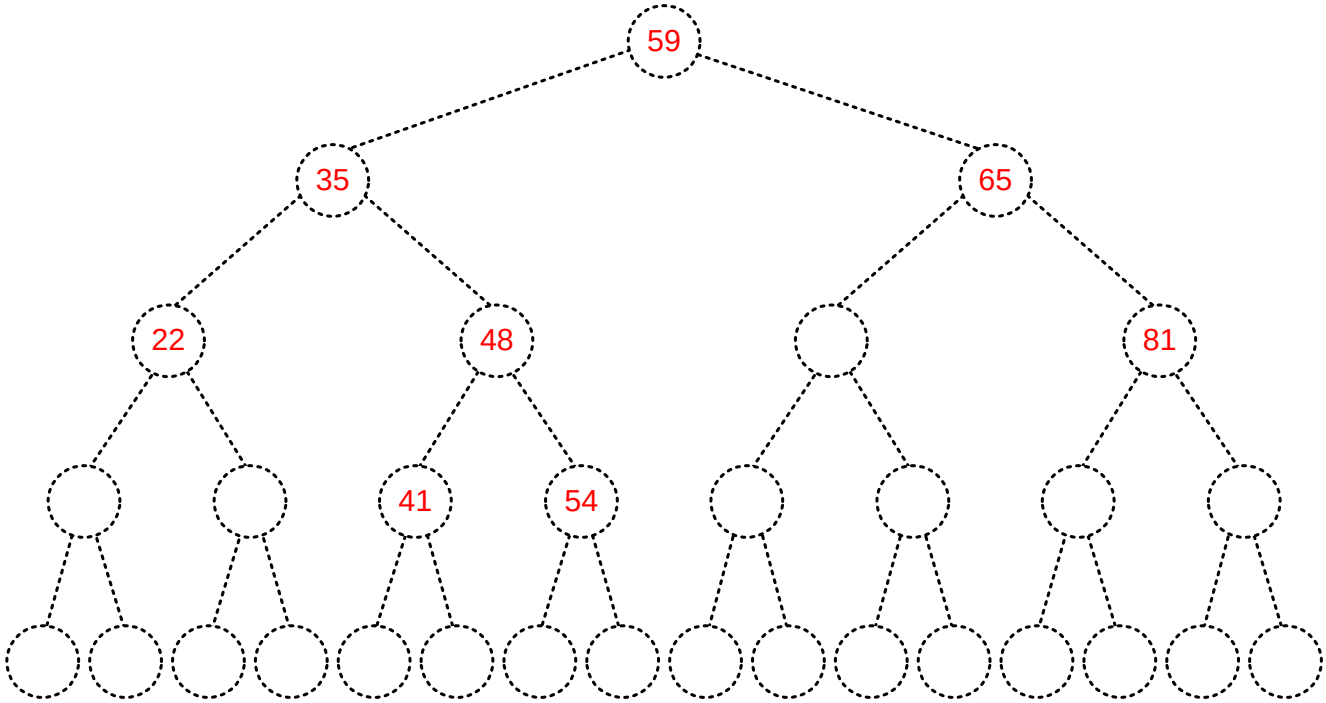
5.1) **(2.5 points)** Insérez 41.



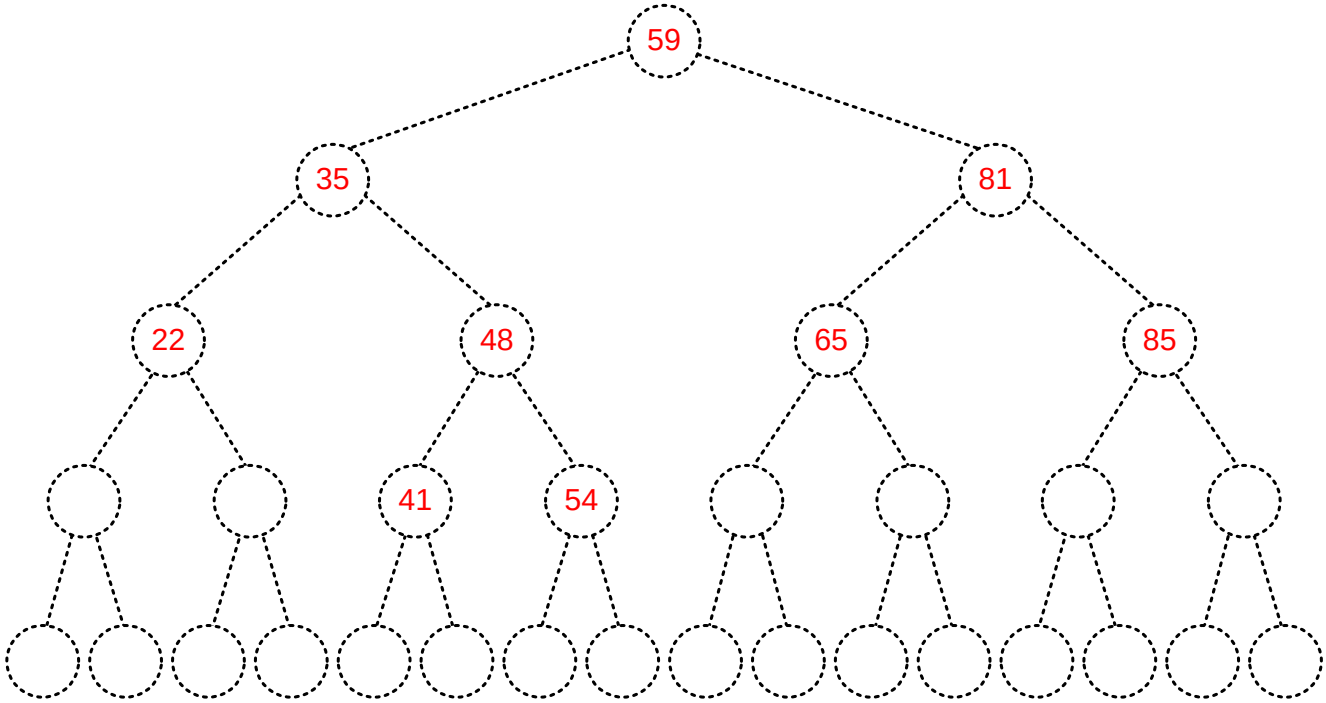
5.2) **(2.5 points)** Insérez 48.



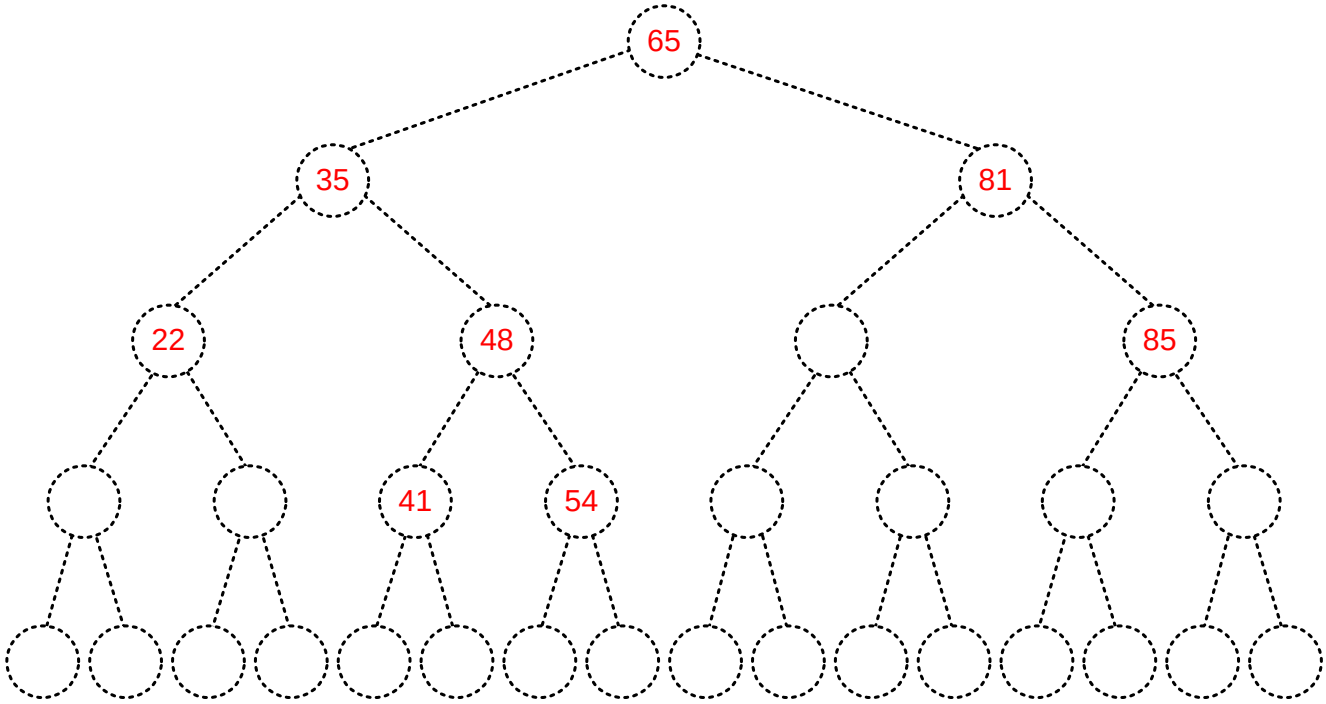
5.3) **(2.5 points)** Insérez 54.



5.4) **(2.5 points)** Insérez 85.



5.5) **(2.5 points)** En partant du résultat de la question 5.4), retirez la racine (utilisez la méthode `remove` d'un arbre binaire de recherche standard telle que vue en classe).

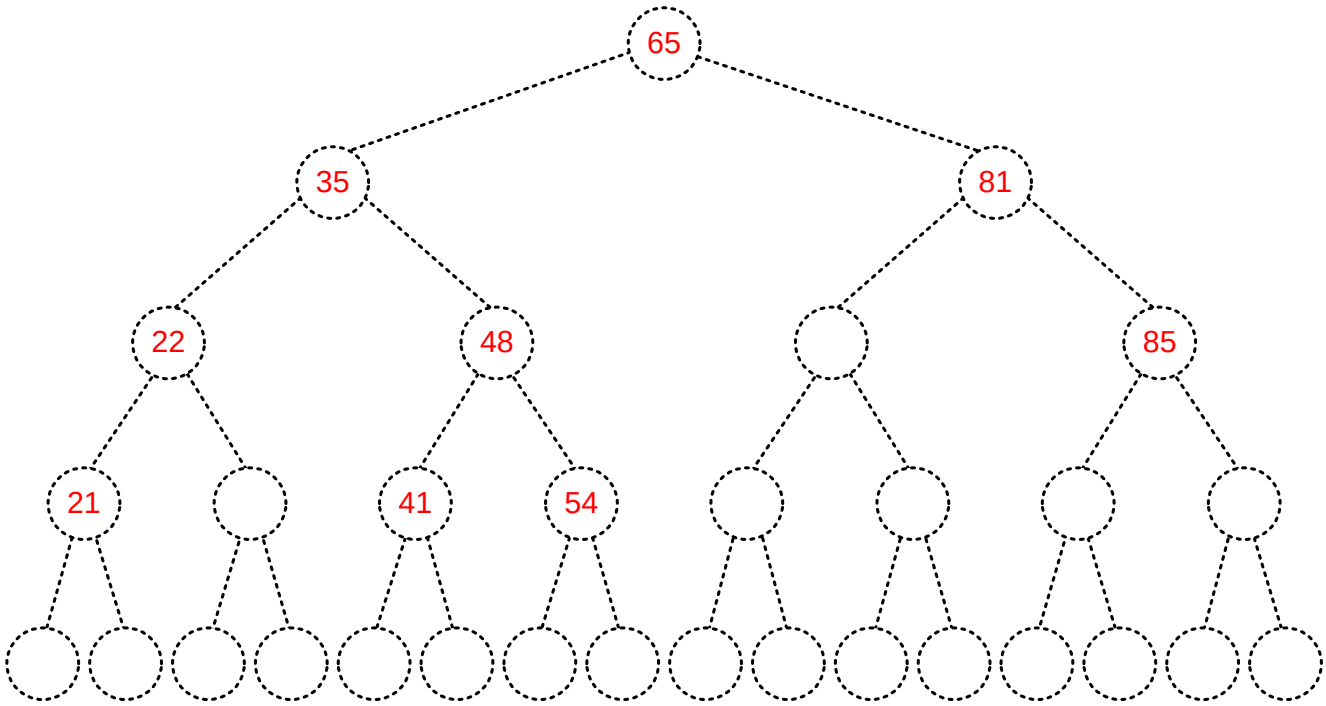


5.6) **(2.5 points)** L'arbre obtenu à la question 5.5) respecte-t-il la structure d'un arbre AVL? Si non, indiquez les nœuds qui violent la règle de l'AVL. Si oui, insérer la clé 21 à cet arbre.

Votre réponse : Oui.

5.6.a) Si votre réponse est NON, énumérer les nœuds violant la règle AVL ci-après :

5.6.b) Si votre réponse est OUI, donnez l'arbre résultant de l'insertion AVL de la clé 21 à l'arbre de votre réponse 5.5) :



Question 6 : Généralités (10 points)

Répondez aux assertions suivantes par « vrai » ou par « faux ». Justifier votre réponse brièvement.

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

Faux. Elle peut se faire en O(n) : exemple merge sort.

6.2) **(2 points)** Si la liste `liste` est une `LinkedList`, le code suivant aura une complexité $O(n^3)$.

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

Faux. Nous avons une complexité $O(n^2)$.

6.3) **(2 points)** Si la liste `list` est une `LinkedList`, le code suivant aura une complexité $O(n)$.

```
public static void retirerCle (List<Integer> liste, int cle )
{
   Iterator<Integer> it = liste.iterator();
   while( it.hasNext() )
      if( it.next().hashCode() == cle )
         it.remove();
}
```

Vrai. La condition if s'exécute en temps constant, idem pour remove. Le tout se fait en O(n).

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

Faux. O(nlg(n)) en tout temps.

6.5) **(2 points)** Un arbre AVL de hauteur h=7 possède au plus 255 nœuds.

Vrai. La formule est $N_{max} = 2^{h+1}-1$. Pour h = 7, $N_{max} = 2^{8}-1 = 255$.

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 ];
   }
}
```