POLYTECHNIQUE MONTRÉAL

Solutionnaire 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**	**20151**
Professeur		***Local***	***Téléphone***
Ettore Merlo, responsable – Tarek Ould Bachir, chargé de cours		**L-2710**	
Jour	***Date***	***Durée***	***Heure***
Vendredi	**20 février 2015**	**2h00**	**15h00**

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 **7** questions sur un total de **19** 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 double Hash(*clé*) = (clé + 2*i*) % N.

1.1) **(9 points)** En vous servant du tableau ci-dessous, donnez l'état de la mémoire d'une table de taille N=13 après l'insertion, dans l'ordre, des clés suivantes:

19, 70, 45, 84, 57, 12.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Entrées						70	19	57	45		84		12

Donnez le détail de vos calculs ci-après:

x = 19: 19%6 = 6
x = 70: 70%13=5
x = 45: 45%13 = 6, collision ; 6 + 2%13 = 8
x = 84: 84%13 = 6, collision ; (6+2)%13 = 8, collision ; (8 + 2)%13 = 10
x = 57: 57%13 = 5, collision ; (5+2)%13 = 7
x = 12: 12%13 = 12

1.2) **(3 points)** Après l'insertion des clés données à la question 1.1), on effectue un appel à `remove(32)` ? Donnez le détail de cet appel. Soyez bref mais précis. Vous pouvez vous aider du code source fourni à l'annexe 1.

32%13 = 6, case non vide, 32 absent ; (6+2)%13 = 8, case non vide, 32 absent ; (8+2)%13 = 10, case non vide, 32 abent; (10 + 2)%13 = 12, case non vide, 32 absent ; (12 + 2)%13 = 1, case vide, 32 n'est pas dans la table de dispersement. Remove retourne sans modifier la structure de donnée.

1.3) **(2 points)** Quelle sera la plus grande valeur prise par *i* (le *i* de Hash(*clé*) = (clé + 2*i*) % N) lors de l'appel `remove(32)`.

i = 4

1.4) **(2 points)** Combien de clés supplémentaires faut-il ajouter à la table de dispersement pour que la fonction `rehash()` soit appelée?

Une de plus

1.5) **(2 points)** Quelle sera la nouvelle taille de la table après l'appel à `rehash()`?

next_prime(2x13) = 29

Question 2 : Tris en $n \log(n)$ (14 points)

On désire exécuter l'algorithme *Quick Sort* pour trier le vecteur ci-après. On considère une valeur *cut-off* de 6. 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	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

2.1) (**2 points**) Donnez l'état du vecteur après la mise à l'écart du pivot lors de la première récursion de QuickSort :

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

2.2) (**3 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**
Valeurs	**1**	**3**	**4**	**5**	**6**	**7**	**2**	**8**	**10**	**11**	**12**	**13**	**14**	**9**	**15**

2.3) **(3 points)** Au total, quel est le nombre de fois que la fonction récursive `quicksort` aura été appelée pour exécuter le tri ? Pour éviter tout ambiguïté, 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, AnyType tmpArray, int left, int right )
```

Votre réponse: **7**

2.4) **(2 points)** Quelle est la plus petite taille de vecteur sur laquelle la fonction `insertionSort` aura été appelée ?

Votre réponse: **1**

2.5) **(2 points)** Quelle est la plus grande taille de vecteur sur laquelle la fonction `insertionSort` aura été appelée ?

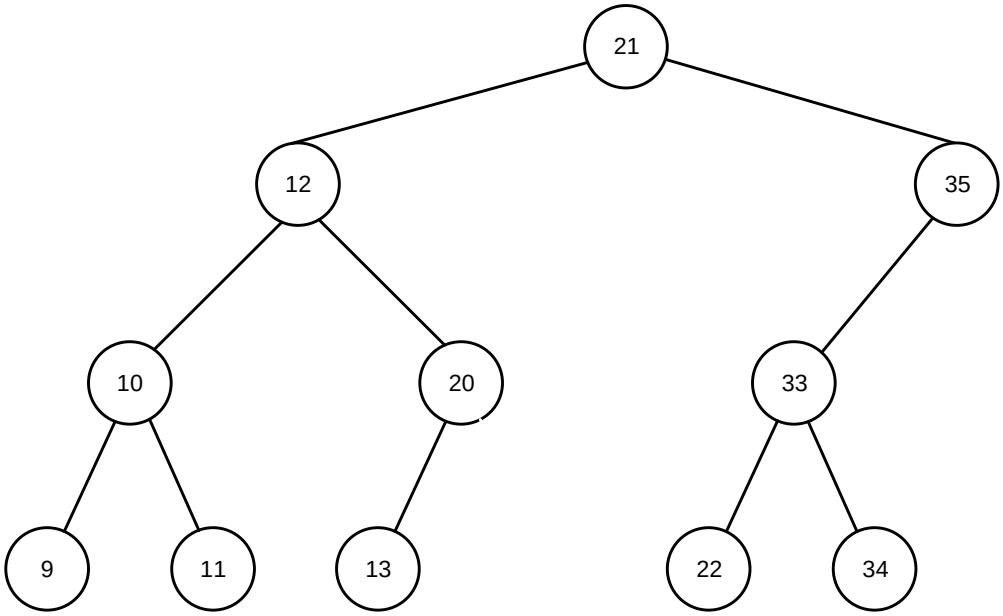
Votre réponse: **5**

2.6) (**2 points**) Reproduisez le vecteur correspondant à la question (2.4) en positionnant ses éléments à leur place (entre les indices `left` et `right` inclusivement) :

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

Question 3 : Parcours d'arbres (13 points)

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.

21, 12, 10, 9, 11, 20, 13, 35, 33, 22, 34

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.

9, 11, 10, 13, 20, 12, 22, 34, 33, 35, 21

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.

9, 10, 11, 12, 13, 20, 21, 22, 33, 34, 35

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.

21, 12, 35, 10, 20, 33, 9, 11, 13, 22, 34

3.5) (**1 point**) Cet arbre respecte-t-il la structure d'un arbre complet ? Justifiez brièvement.

Non, 20 et 35 n'ont pas d'enfant à droite.

3.6) (**1 point**) Cet arbre respecte-t-il la structure d'un arbre binaire de recherche ? Justifiez brièvement.

Oui, les clés à la gauche d'un nœud sont plus petites que sa clé, celles à sa droite sont plus grande.

3.7) (**1 point**) Cet arbre respecte-t-il la structure d'un AVL ? Justifiez brièvement.

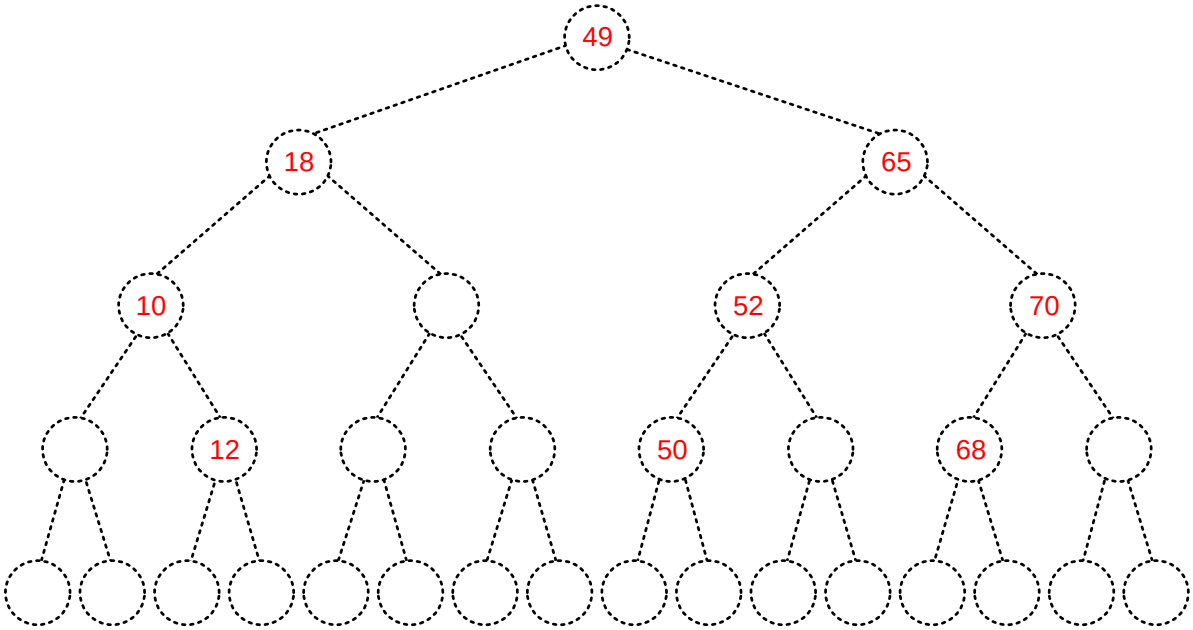
Non, le nœud 35 a un arbre de hauteur h_L=1 à sa gauche, et de hauteur h_R = -1 à sa droite : |hG-hR| > 1

Question 4 : Arbres binaire de recherche (16 points)

4.1) **(4 points)** Si l'affichage par niveaux de l'arbre binaire de recherche donne :

49, 18, 65, 10, 52, 70, 12, 50, 68

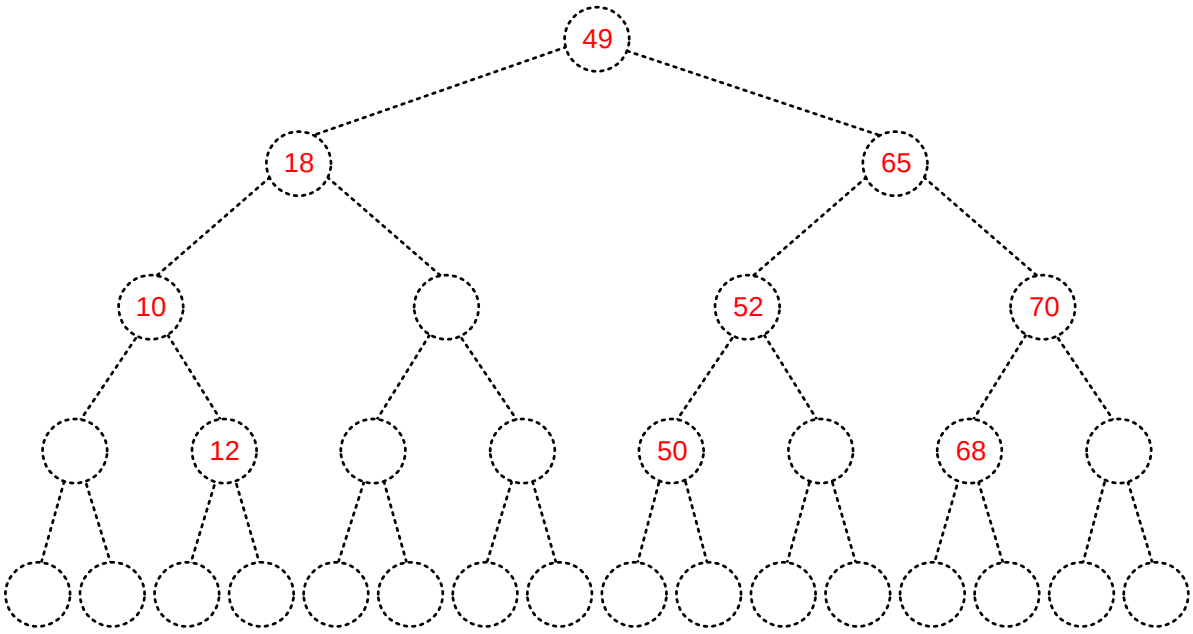
Donnez la représentation graphique de l'arbre.



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

49, 18, 10, 12, 65, 52, 50, 70, 68

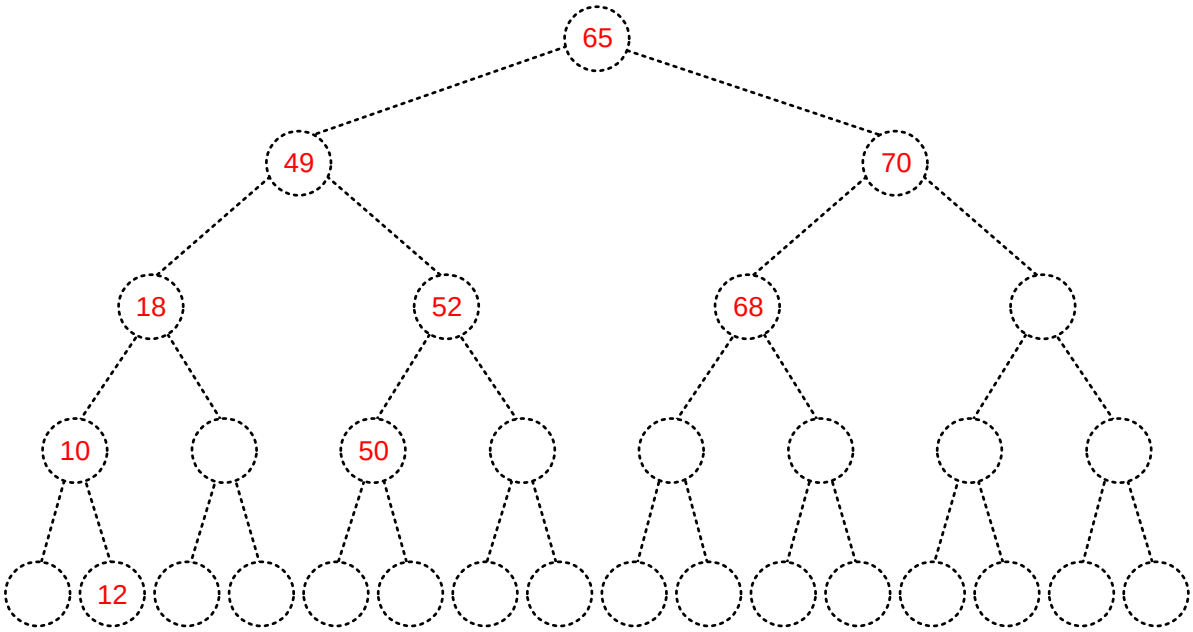
Donnez la représentation graphique de l'arbre.



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

12, 10, 18, 50, 52, 49, 68, 70, 65

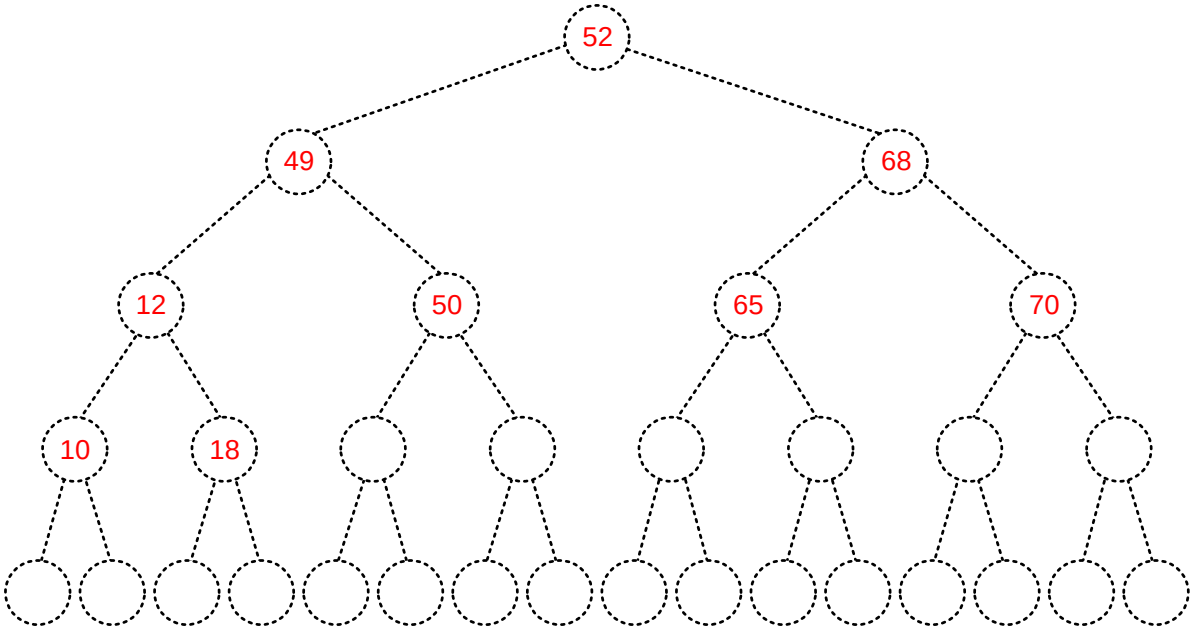
Donnez la représentation graphique de l'arbre.



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

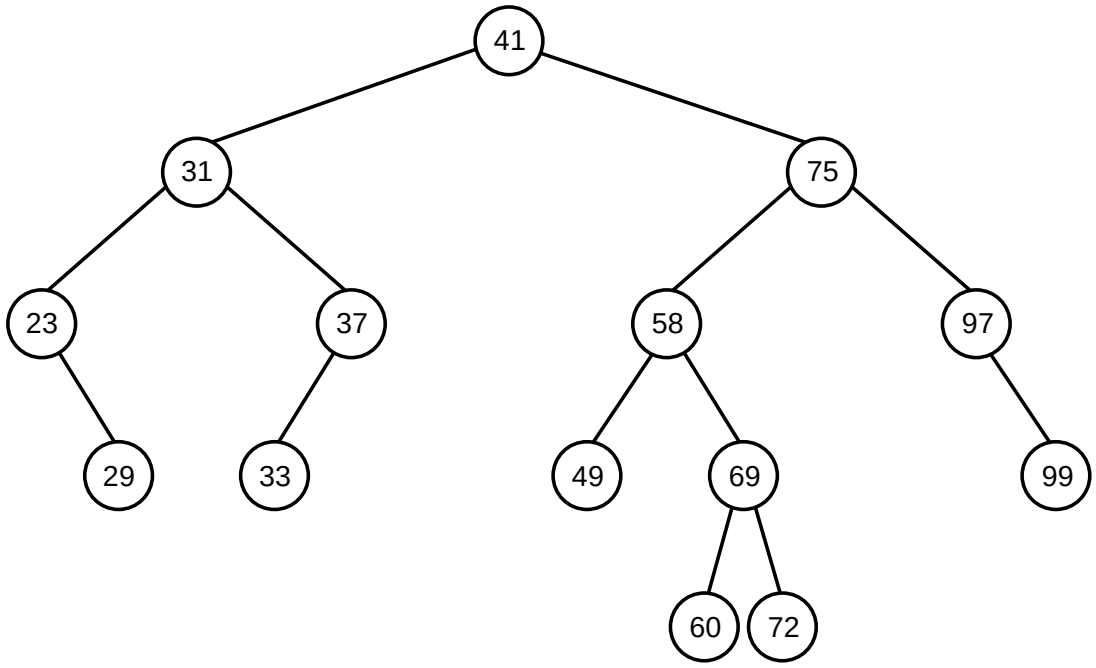
10, 12, 18, 49, 50, 52, 65, 68, 70

Donnez la représentation graphique de l'arbre, sachant qu'il s'agit également d'un arbre complet.



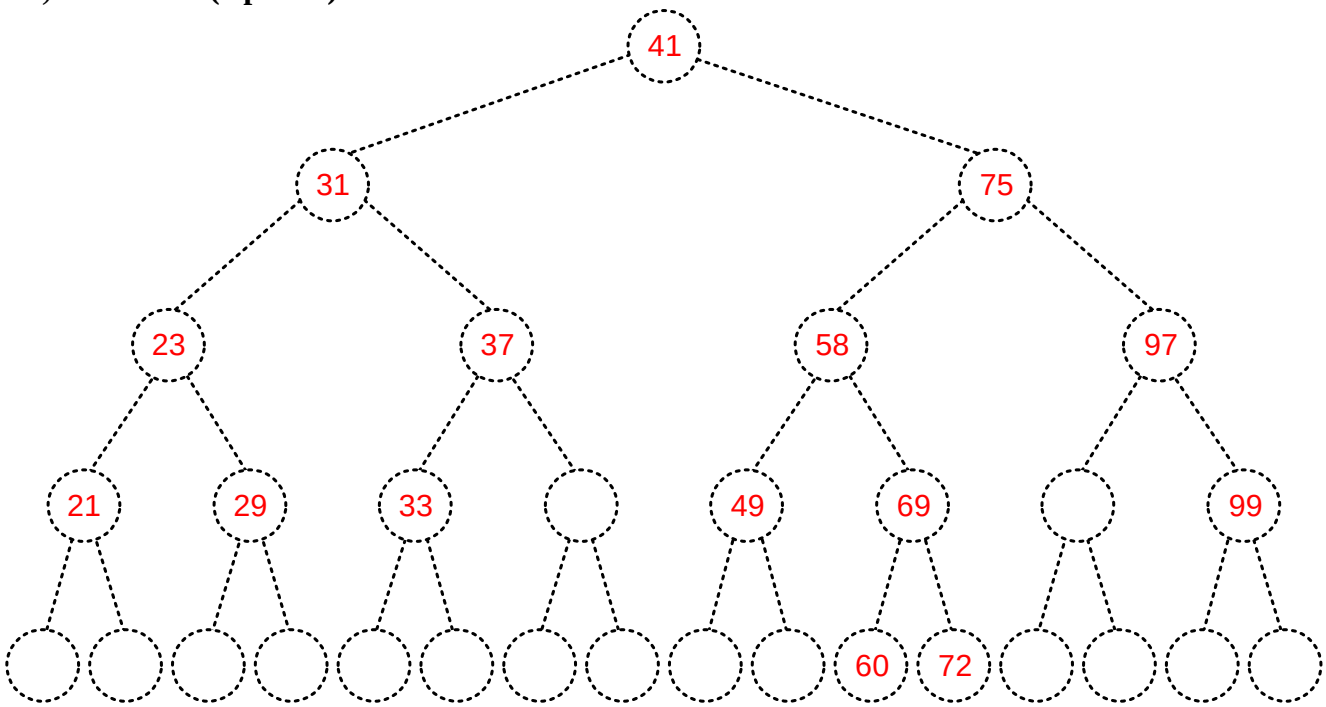
Question 5 : Arbre binaire de recherche de type AVL (19 points)

En partant de l'arbre AVL suivant :

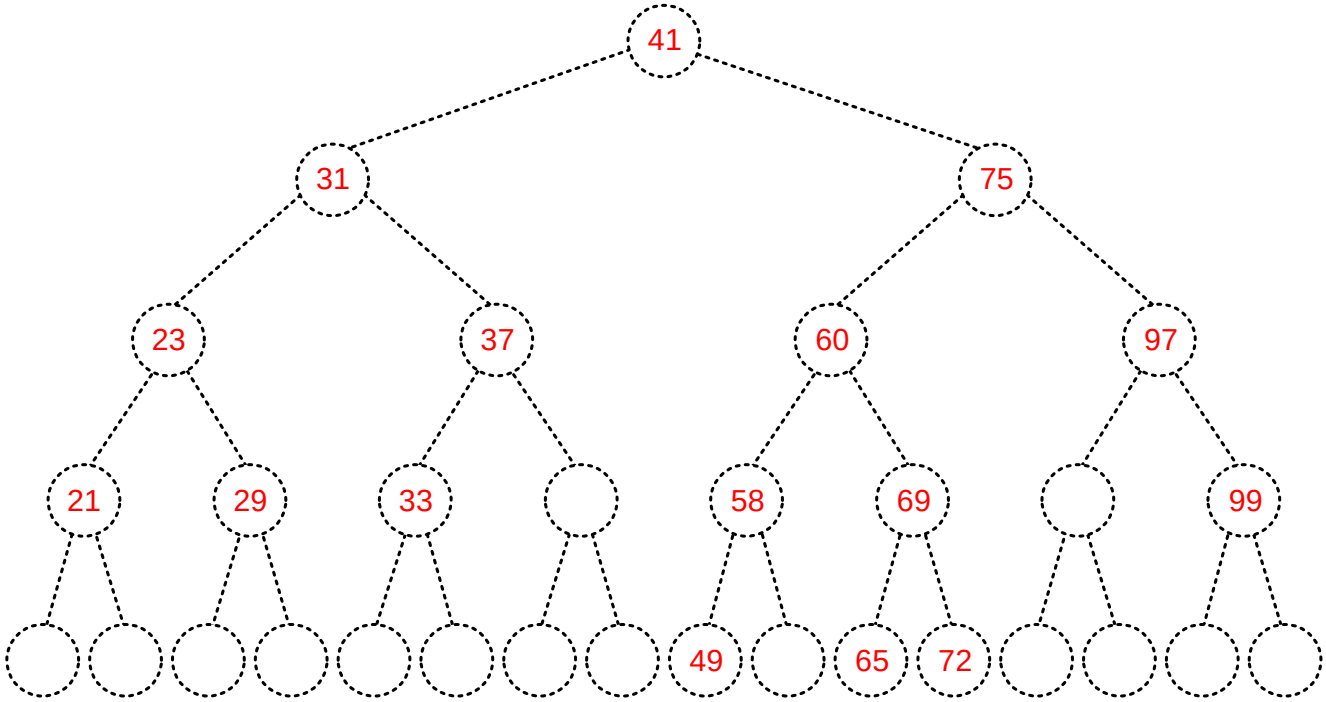


Insérez dans l'ordre les clés suivantes : 21, 65, 74, 42

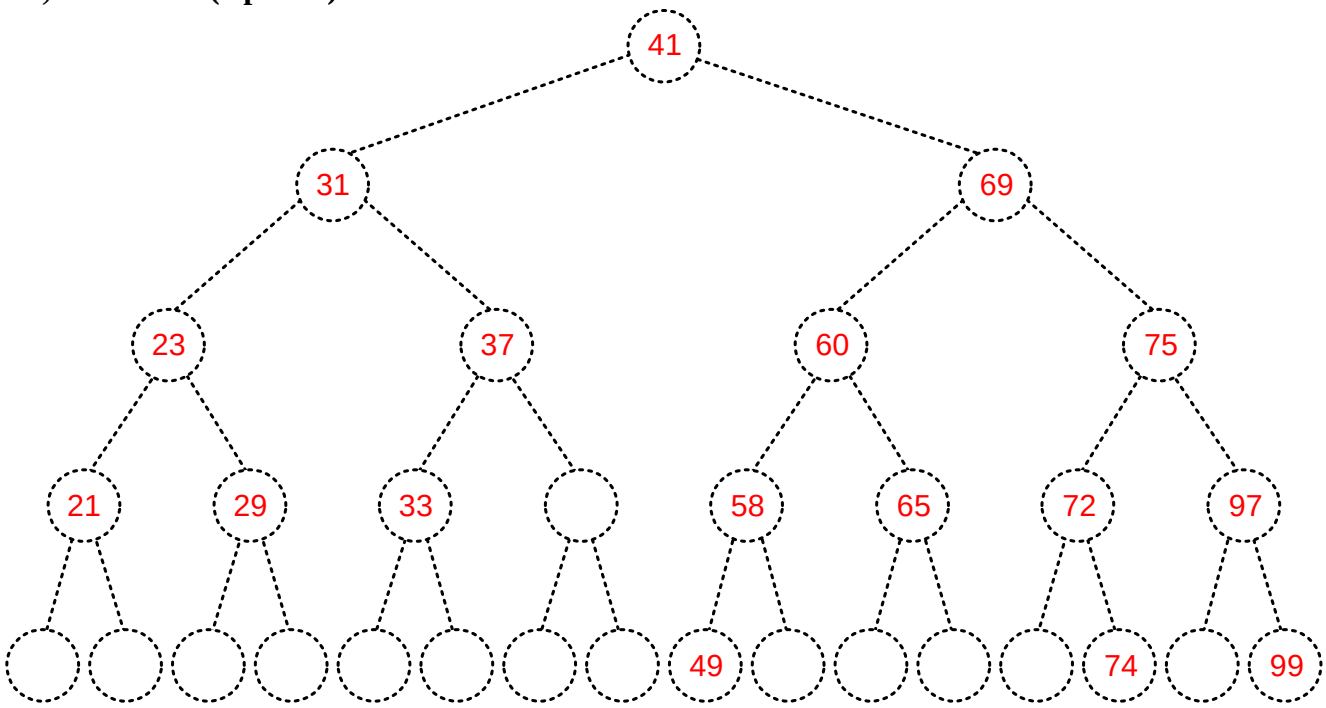
5.1) Insérez 21. **(3 points)**



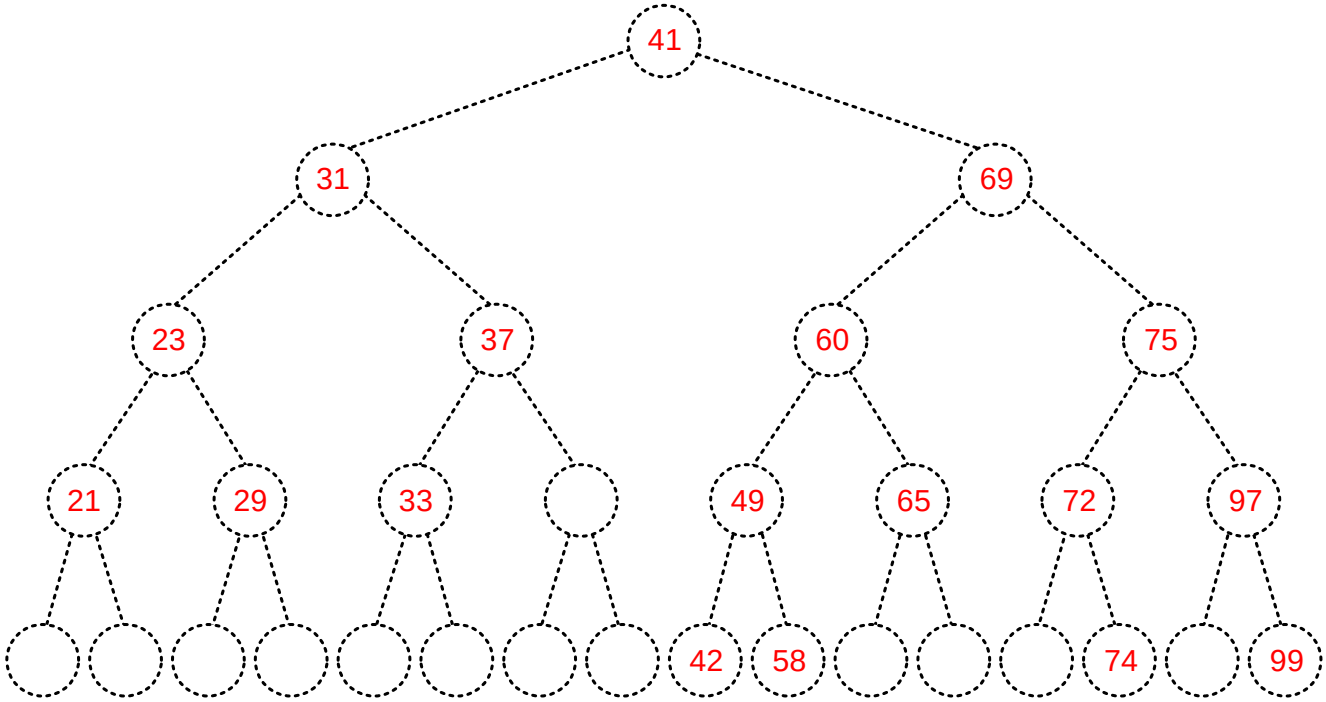
5.2) Insérez 65. **(4 points)**



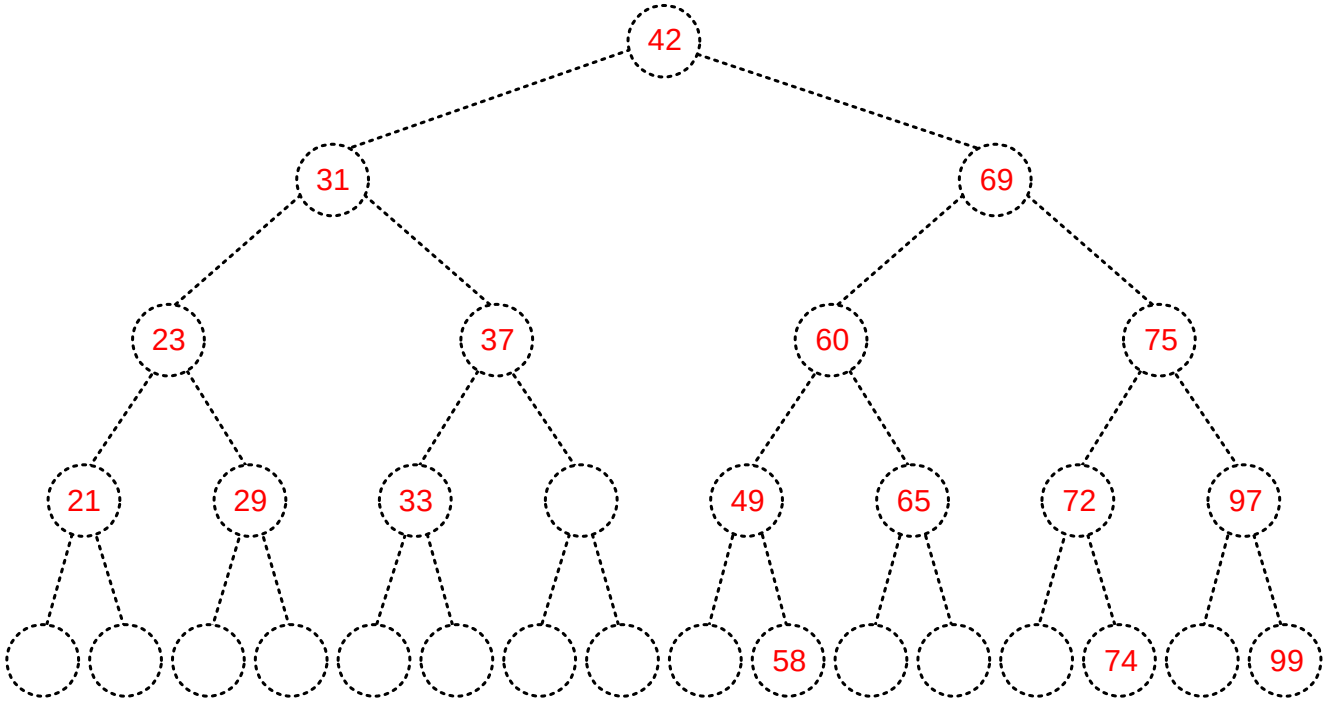
5.3) Insérez 74. **(4 points)**



5.4) Insérez 42. **(4 points)**



5.5) En partant du résultat de la question 5.4), retirez la racine. **(4 points)**



Question 6 : Généralités (20 points)

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

a) La signature suivante est bonne pour implémenter un itérateur sur la liste `MaListe`. **(2 pts)**

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

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

Faux : la classe implémentant l'interface Iterator ne devrait pas être static

b) L'algorithme QuickSort a une complexité O(n log(n)) en pire cas **(2 pts)**.

Faux : $O(n^2)$

c) L'algorithme MergeSorte a une complexité O(n) en meilleur cas **(2 pts)**.

Faux : O(nlog(n)) en tout temps

d) Une table de dispersement utilisant une résolution de collision par sondage quadratique doit toujours avoir une taille qui est un nombre premier **(2 pts)**.

Vrai : sinon risque qu'une entrée soit impossible à insérer

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

Faux : il faut également que le facteur de compression soit au plus de 50%

f) L'opération de retrait usuelle, telle qu'effectuée sur un arbre binaire de recherche standard, effectuée sur un arbre AVL ne produit pas un arbre AVL **(2 pts)**.

Faux : elle peut produire un AVL, voir l'exemple de la question 5.5, dire qu'elle ne produit pas nécessairement un AVL est plus juste.

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

Faux: on peut avoir O(nlog(n))

h) Un `remove()` sur une liste pat tableau s'effectue en O(n^2). **(2 points)**.

Faux : O(n)

i) Un arbre AVL de hauteur h=9 possède au plus 143 nœuds. **(2 points)**.

Faux : Il possède au plus 1023 noeuds

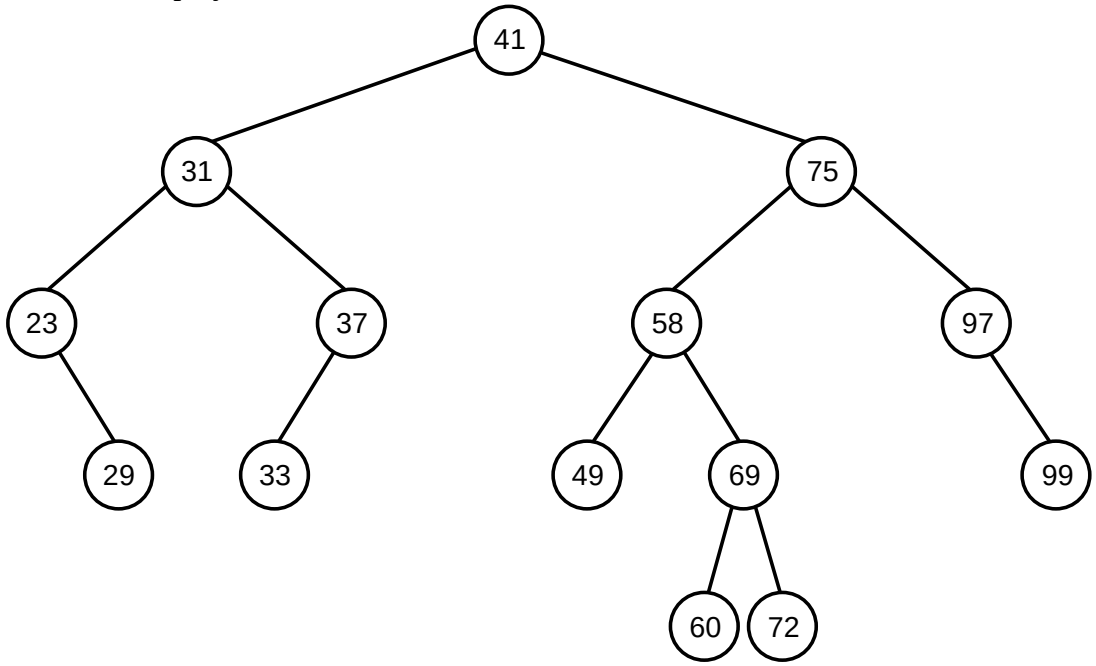
j) Un arbre AVL ayant 87 nœuds a au plus une hauteur de h=7. **(2 points)**.

Vrai : l'AVL le moins peuplé de hauteur h=8 possède 88 nœuds

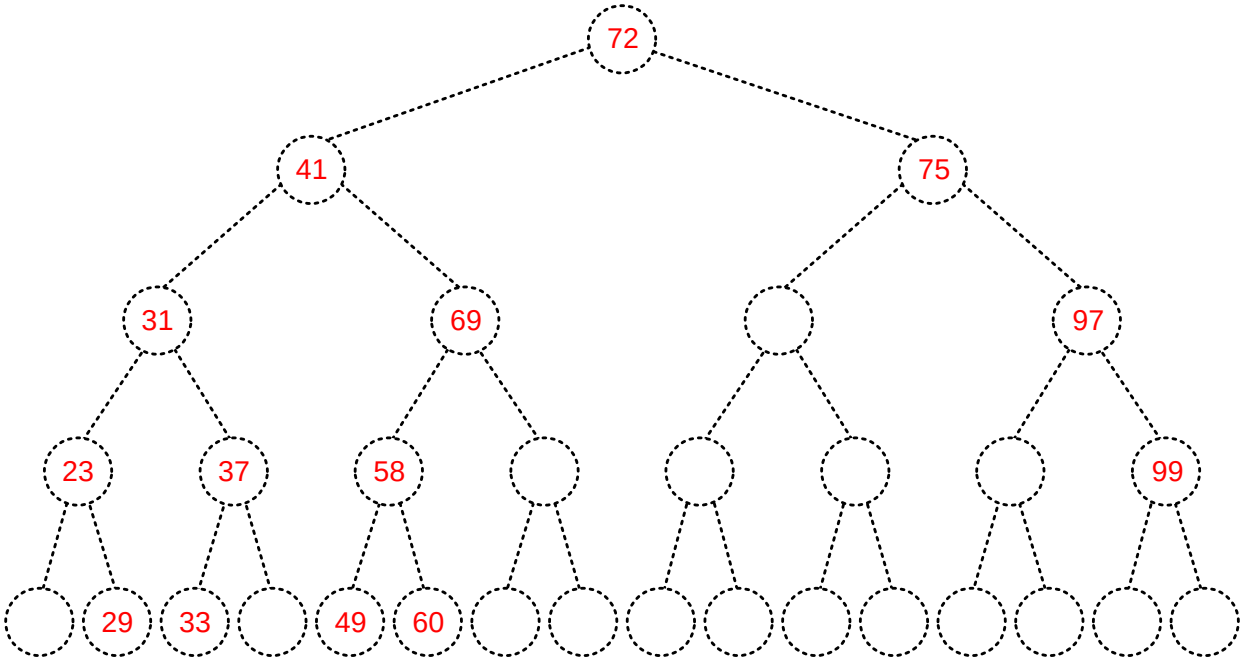
Question 7 : Arbre binaire de recherche de type Splay (5 points)

Cette question est en bonus. En réussissant cet exercice, le total de points obtenus pour l'intra peut dépasser les 100%.

En partant de l'arbre Splay suivant :



Effectuez un `get(72)`.



Annexe 1

```
public class LinearProbingHashTable<AnyType>
{
    public LinearProbingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    public LinearProbingHashTable( 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( )
    {
        System.out.println("DEBUG: 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 currentPos = myhash( x );

        while( array[ currentPos ] != null &&
                !array[ currentPos ].element.equals( x ) )
        {
            currentPos += 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 )
{
    int currentPos = findPos( x );
    return isActive( currentPos );
}

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 )
    {
        this( e, true );
    }

    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 == 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;
    }

    public static void main( String [ ] args )
    {
        LinearProbingHashTable<Integer> H = new LinearProbingHashTable<Integer>( );

        Integer[] values = {19, 70, 45, 84, 57, 12};

        for( int item : values )
            H.insert( item );

        H.remove(32);
    }
}
```

Annexe 2

```
public final class Sort
{
    public static <AnyType extends Comparable<? super AnyType>>
    void quicksort( AnyType [ ] a )
    {
        quicksort( a, 0, a.length - 1 );
    }

    private static final int CUTOFF = 6;

   public static <AnyType> void swapReferences(AnyType [ ] a, int index1, int index2)
    {
        AnyType tmp = a[ index1 ];
        a[ index1 ] = a[ index2 ];
        a[ index2 ] = tmp;
    }

    private static <AnyType extends Comparable<? super AnyType>>
    void quicksort( AnyType [ ] a, int left, int right )
    {
        System.out.println("DEBUG: call to quicksort");

       if( left + CUTOFF <= right )
        {
            AnyType pivot = median3( a, left, right );

                // Begin partitioning
            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 );

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

```
    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;
        }
    }

    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 ];
    }

    public static void main( String [ ] args )
    {
        Integer [ ] a = new Integer[ 15 ];
        for( int i = 0; i < a.length; i++ )
            a[ i ] = 15 - i;

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

        System.out.println();

        Sort.quicksort( a );

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

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