***Indications :***

* La classe Bucket est caractérisé par :
  + - Une liste des enregistrement appartenant a le même groupe
    - Un attribut «ShouldSwap» qui marque si les enregistrement de la bucket sont successif ou non
* La classe nœud est caractérisé par :
  + - Un attribut « parent » de type Nœud
    - Un attribut « pathWeight » qui contient le poids pour passer du nœud en question vers son parent
    - Une Liste des fils « childrenNodes »

***Algorithme :*** *Anonymisation*

***Entré :***

bucketList :list des Buckets

hierarchie : nœud

***Variables :***

anonymousCode :chaine <- ‘’ ‘’ ;

targetNode :nœud ;

helper : chaine <- ‘’ ‘’ ;

helperNode :nœud ;

helperBucketList :list<bucket> ;

tempBkt1, tempBkt2 :bucket ;

***Début :***

**Pour chaque** bucket dans bucketList **faire :**

**Pour chaque** enregistrement dans bucket **faire :**

anonymousCode <- generateDiseaseCode(enregistrement.disease,hierarchie) ; // faire appel a la fonction qui génère le code anonymisé de la maladie

enregistrement.setDiseaseCode( anonymousCode) ; // attribuer le code anonymisé de la maladie a l’enregistrement en question

**Fin pour**

**Fin pour ;**

Appeler bucketClassifier (bucketList); // appel a la procédure qui calcifie les bucket qui doivent être participer a la permutation de celle qui ne doivent pas

Fonction traverse ( Tgdisease :chaine ,node :Node) : Nœud

**Si** node.getDisease() ==TgDisease  **faire**: // condition d’arrêt

renvoyer node ;

**Sinon :**

**Pour chaque** neoud dans node.getChildren() **faire :**

helperNode= traverse(TgDisease,neoud ) ;// appel récursive de la fonction

**si** helperNode !=null **faire**:

renvoyer helperNode ;

**fin si ;**

**fin pour ;**

**Fin si ;**

Fin Fonction ;

**Procédure** bucketClassifier( bucketList :List) :

**Pour chaque** buckt **dans** bucketList  **faire :**

**Si** buckt.getRecords() >1 **faire :**

Prefix <- buckt.getRecords().get(0).getAnonimizedCode().subString(0,3) // on recupere le prefix du premier element

Buckt.setShouldSwap(true);// on suppose que les element du buckt doivent etre permuté

**Pour chaque** i=0 **jusqu’à** buckt.getRecords.size()   **faire :**

// si les elements de la buckets n’ont pas le meme prefix ,dans ce cas

Les elements de la bucket ne doivent pas etre permutés

**Si !** buckt.getRecords.get(i). getAnonimizedCode().startsWith(prefix) **faire**:

Buckt.setShouldSwap(false);

**fin si ;**

**fin pour ;**

**fin si ;**

**fin pour ;**

fin **Procédure**;

**Procédure** recordsShuffler ( bucketList :List, opt :chaine) :

// on recupere les buckets qui voivent participer a la permutation

**Pour chaque** buckt **dans** bucketList  **faire :**

**Si !** buckt.getShouldSwap = true **faire**:

helperBucketList.add(buckt) :

**fin si ;**

**fin pour ;**

//si il y a une seule bucket a permuter, on la permute avec les éléments d’une autre choisi aléatoirement

**Si** helperBucketList.getSize() =1 **faire**:

tempBkt1 <- getBuckt(bucketList ) ;

tempBkt2 <- helperBucketList.get(0);

appler swap( tempBkt1, tempBkt2,opt); // appel a la procédure qui fait la permutation

**fin si ;**

// si il y a un nomber paire des bucket a permuter ,on les permute deux par deux

**sinonsi** helperBucketList.getSize() % 2 = 0 **faire:**

**pour** i=0 ;i< helperBucketList.getSize();i=i+2 **faire :**

// dans le code de l’application on vérifie de telle sort que les deux bucket ne contiennent pas des éléments appartenant au même hiérarchie

tempBkt1 <- helperBucketList.get(i);

tempBkt2 <- helperBucketList.get(i+1);

appler swap( tempBkt1, tempBkt2,opt);

**fi pour ;**

**fin sinonsi ;**

// si il y a un nomber unpaire different de 1

**sinonsi** helperBucketList.getSize() % 2 =1 && workingbuckets.size() != 1**faire:**

**pour** i=0 ;i< helperBucketList.getSize();i=i+2 **faire :**

// dans le code de l’application on vérifie de telle sort que les deux bucket ne contiennent pas des éléments appartenant au même hiérarchie

tempBkt1 <- helperBucketList.get(i);

tempBkt2 <- helperBucketList.get(i+1);

appler swap( tempBkt1, tempBkt2,opt);

**fi pour ;**

// la bucket qui reste on la permute avec une l’une des bucket resultat de la premiere operation

tempBkt1 <- getBuckt(result ) ; // result est une liste qui contient les buckets resultant de la premiere permutation ,

tempBkt2 <- helperBucketList.get(helperBucketList.size() -1);

appler swap( tempBkt1, tempBkt2,opt);

**fin sinonsi ;**

fin **Procédure**;

Fonction generateDiseaseCode( disease :chaine,hierarchy :node) :chaine

targetNode <- traverse(disease :chaine, hierarchy : node) ; // trouver le nœud dans la hiérarchie qui correspond a la maladie

**Tant que** targetNode.getParent  != null **faire :** // arrêter si le sommet de l’arbre est atteint

Helper <- targetNode.getPathWeight(); // chaque nœud a un poids pour arriver a son parent

targetNode <- targetNode.getParent();

**Fin tantque ;**

renvoyer helper; // retourner l’inverse de la chaine contenant la concaténation des poids

fin Fonction ;

**Procédure** swap (b1 :buckt,b2 :bucket : operation :chaine) :

**Si** chaine=’minTOmin’ **faire :**

Temp=b1.getRecords.get(0) ;

b1.getRecords.set(0, b2.getRecords.get(0));

b2.getRecords.set(0,temp);

**fin si ;**

**sinonsi** chaine=’maxTOmax’ **faire :**

Temp=b1.getRecords.get(b1.size()-1) ;

b1.getRecords.set(b1.size()-1, b2.getRecords.get(0));

b2.getRecords.set(b1.size()-1,temp);

**fin si ;**

// les autres options de permutation

fin **Procédure**;

***Fin :***

***Sortie :*** anonymizedList