Stage ETE 2015

Description du système

F-PHT
Un système d'index de filtres de Bloom pour la recherche d'information par mots clés

Réalisé par **DOAN** Cao Sang
Encadrant : M. **MAKPANGOU** Mesaac, Regal

1 Juillet 2015

Table des matières

Chapitre 1

Vue globale

1.1 Prefix Hash Tree (PHT)

Un arbre préfixe est un arbre numérique ordonné qui est utilisé pour stocker une table associative où les clés sont généralement des chaînes de caractères. Contrairement à un arbre binaire de recherche, aucun nœud dans le trie ne stocke la chaîne à laquelle il est associé. C'est la position du nœud dans l'arbre qui détermine la chaîne correspondante [1].

Pour tout nœud, ses descendants ont en commun le même préfixe. La racine est associée à la chaîne vide. Des valeurs ne sont pas attribuées à chaque nœud, mais uniquement aux feuilles et à certains nœuds internes se trouvant à une position qui désigne l'intégralité d'une chaîne correspondante à une clé.

Pour faire une recherche d'une valeur associée à une clé, au départ, on se situe à la racine de l'arbre, en prenant le premier élément de la clé de la requête, on trouve le chemin étiqueté par cet élément, s'il n'existe pas, on est sûr que cette clé n'est pas dans l'arbre. Dès que l'on trouve le chemin, on arrive sur le bon nœud et continue en prenant le deuxième élément de la clé de requête, on applique cette méthode jusqu'à quand on trouve cette clé et se termine sur une feuille.

1. Wikipédia

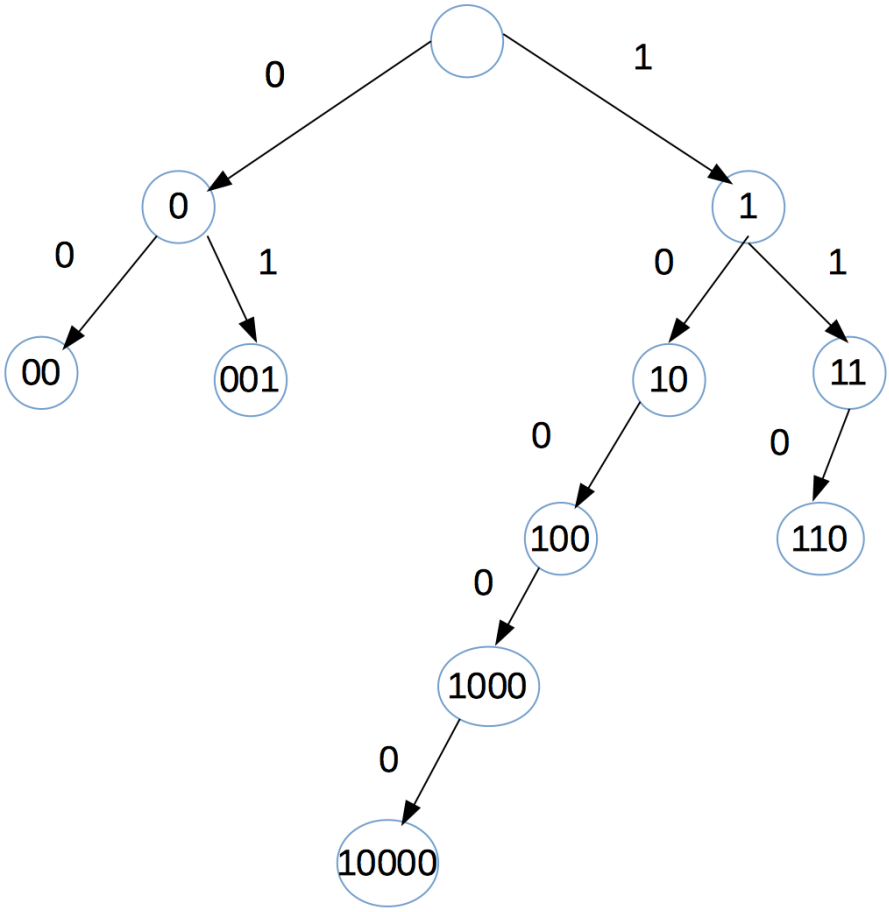

FIGURE 1.1 – Arbre de préfixe

1.2 F-PHT

F-PHT est un arbre préfixe de "multibit". Pour stocker une clé, on suit le chemin étiqueté par les fragments successifs de cette clé (dans l'ordre de leur rang) jusqu'à ce qu'on arrive sur une feuille pouvant stocker cette clé.

Un fragment est un morceau d'une clé (sous-ensemble de bits). Si on considère les clés de taille m, le système découpe chaque clé en f fragments de taille identique. Par convention, ces fragments sont numérotés de *0* à *f-1*.

15															0
1	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0

TABLE 1.1 – Exemple le filtre de Bloom

La table 1.1 représente une clé de taille $m = 16$ bits, elle est ensuite découpée en $f = 4$ fragments. Le premier fragment est la suite de bits *"1000"* qui se trouve les plus à gauche de cette clé. Le dernier est *"1010"*.

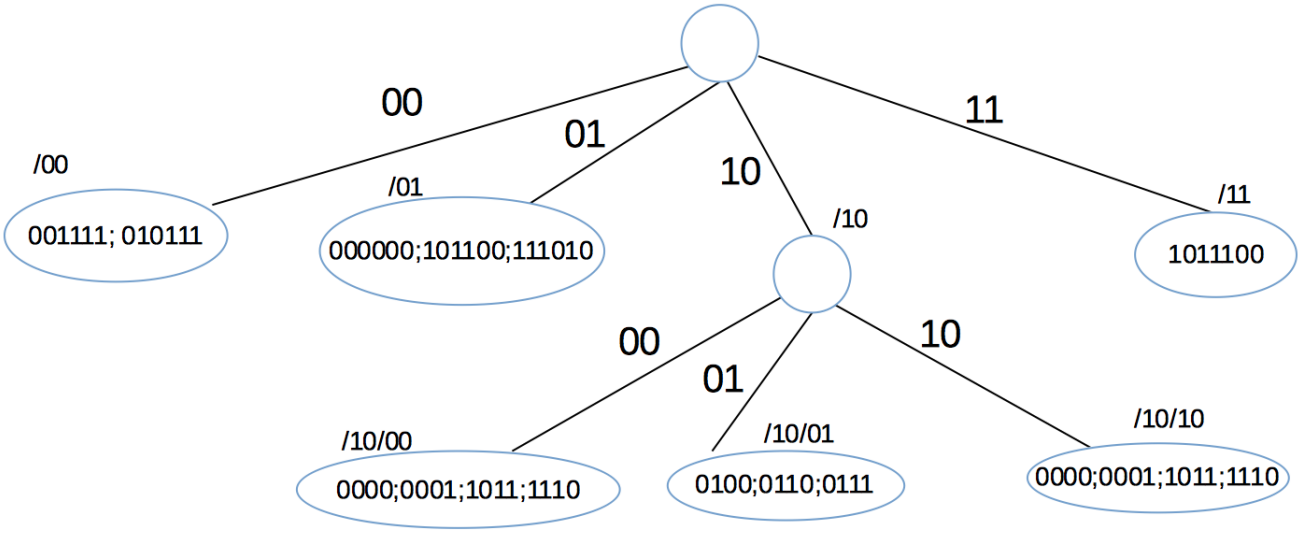

FIGURE 1.2 – Arbre F-PHT

La figure 1.2 représente l'arbre F-PHT qui stocke une clé de taille $m = 8$ bits, la taille d'un fragment est de 2 bits. Au dessus de chaque nœud (interne ou feuille) se trouve son identifiant unique et il caractérise également le chemin unique qui conduit vers ce nœud.

Chapitre 2

Système d'indexation

2.1 Objectif

Les clés indexées sont des filtres de Bloom de taille m. L'objectif du système est d'une part de stocker les filtres de Bloom donnés par les utilisateurs, d'autre part de permettre de rechercher tous les filtres de Bloom (stockés dans le système) qui sont sur-ensemble d'un filtre de Bloom caractérisant des mots clés d'une requête.

2.2 Architecture du système d'indexation

Le système d'indexation est réalisé par un ensemble de serveurs d'index répartis sur plusieurs machines. Chaque serveur gère un ou plusieurs nœuds de l'arbre F-PHT. Ce système permet de créer plusieurs index, chaque index maintient un F-PHT qui stocke les clés.

On suppose qu'il y a un serveur central connu qui gère tous les index créés. Il utilise une table des index, chaque entrée sert à sauvegarder l'identifiant de l'index *indexID* et le serveur *rootNodeID* qui héberge sa racine. Lorsqu'un utilisateur veut créer/supprimer un index, il demande le serveur central d'ajout/suppression une entrée dans la table des index.

2.3 API du système d'indexation

Ce système offre aux utilisateurs des services suivants :

createIndex() : créer un nouveau index et rend aux l'utilisateur l'identifiant d'index créé *indexID*.

removeIndex(indexID) : supprime un index identifié par *indexID*.

add(indexID, key) : ajoute dans l'index identifié par *indexID* une clé *key*.

remove(indexID, key) : supprime une clé *key* dans l'index identifié par *indexID*.

search(indexID, key_req) : recherche dans l'index identifié par *indexID* des sur-ensembles qui satisfont la requête *key_req*.

Chapitre 3

Fonctionnement du système

3.1 Données de chaque nœud dans l'arbre

Identifiant d'un nœud caractérise le nœud. Cet identifiant est calculé grâce au chemin vers ce nœud, et ce chemin est unique.

Niveau d'un nœud (rang) est hauteur dans l'arbre F-PHT.

Table de routage (localRoute) sert :
- soit à router chaque requête vers le nœud fils responsable de la traiter.
- soit à déterminer l'adresse du conteneur local. Ce conteneur stocke au maximum γ éléments.

3.2 Protocole

3.2.1 Ajout d'un filtre de Bloom dans F-PHT

Lorsqu'un nœud reçoit une requête d'ajout d'un filtre de Bloom (c'est-à-dire d'une clé), il effectue deux actions :

1. Premièrement, il détermine la valeur du fragment numéro i de la clé à ajoutée, avec i correspondant au niveau de ce nœud dans l'arbre F-PHT.
2. Une fois le fragment de la clé de rang i déterminé, il consulte l'entrée de la table de routage *localRoute* associée à cette valeur. Trois cas sont à distinguer :
   (a) Si l'entrée est nulle (aucune clé ayant la même valeur pour le fragment de rang i a été ajoutée auparavant), le nœud crée un conteneur local, ajoute son adresse dans cette entrée et stocke la clé reçue dans ce conteneur.
   (b) Si l'entrée associée à la valeur du fragment de rang i contient l'identifiant d'un nœud fils, on transmet la requête vers ce fils.
   (c) Si l'entrée contient l'adresse d'un conteneur local, on ajoute la clé reçue dans ce conteneur. Si après ajout, le nombre d'éléments stockés dépasse le seuil fixé, le nœud crée un nœud fils, lui transmet le conteneur, et met à jour l'entrée correspondante l'adresse de son nouveau fils.

3.2.2 Recherche des filtres de Bloom dans F-PHT

Lorsqu'un nœud reçoit une requête de recherche des sur-ensembles d'un filtre de Bloom, il effectue les étapes suivantes :

1. D'abord, il détermine la valeur du fragment numéro i de la clé de requête, avec i correspondant au niveau de ce nœud dans l'arbre F-PHT.

2. Ensuite, il consulte toutes les entrées de la table de routage *localRoute* contiennent la valeur du fragment déterminée précédente. Pour chaque entrée, il y a 3 cas à traiter :
   (a) Si l'entrée est nulle (aucune clé contenant cette valeur du fragment de rang i), le nœud continue.
   (b) Si l'entrée associée à la valeur du fragment de rang i contient l'identifiant d'un nœud fils, on transmet la requête vers ce fils.
   (c) Si l'entrée contient l'adresse d'un conteneur local, on collecte tous les filtres qui contiennent la requête. L'ensemble des filtres collectés est retourné comme réponse.
3. Puis, il attend toutes les réponses des fils auxquels il ont transmis la requête. Une fois, il reçoit toutes les réponses et réunit avec son ensemble des résultats, il transmet le résultat au père.

3.2.3 Suppression d'un filtre de Bloom dans F-PHT

Les étapes de la suppression d'un filtre de Bloom dans F-PHT effectuées par un nœud sont :

1. Il détermine la valeur du fragment numéro i de la clé de requête, avec i correspondant au niveau de ce nœud dans l'arbre F-PHT.
2. Ensuite, il consulte l'entrée de la table de routage *localRoute* associée à cette valeur. Trois cas sont à distinguer :
   (a) Si l'entrée est null (aucune clé ayant la même valeur pour le frament numéro i a été ajoutée auparavant), le nœud rejette la requête.
   (b) Si l'entrée associée à la valeur du fragment de rang i contient l'identifiant d'un nœud fils, on transmet la requête vers ce fils.
   (c) Si l'entrée contient l'adresse d'un conteneur local, on supprime la clé de requête dans ce conteneur. Si après suppression, ce conteneur devient vide, on supprime ce conteneur et cette entrée dans la table *localRoute*. Si cette table devient vide, il notifie le père de la suppression de son fils et demande le système de supprimer ce nœud. Si le père reçoit la notification de son fils, il vide l'entrée où stocke son adresse. En plus, si la table de routage *localRoute* de son père devient aussi vide, il fait la même procédure comme son fils : notifie son père, demande la suppression du système.

Chapitre 4

Implémentation du système (Etape 1)

Ce système est implémenté en Java.

4.1 Plateforme de simulation

Cette plateforme est destinée à simuler le système d'indexation. Elle crée un index, ensuite ajoute des données dans cet index. A la fin, une suite de requête de recherche des données est lancée pour tester le fonctionnement du système. Elle peut également simuler la suppression des données dans le système.

L'application interagit avec le système via l'API.

4.2 Serveur

Le serveur s'occupe de la communication entre le client et le système via une interface d'utilisateur. Le client donc peut créer des index, ajouter des filtres, rechercher et supprimer les filtres et les index qui lui appartiennent.

En plus, le serveur est aussi le lieu où on stocke les nœuds du système. Dans le premier temps, il y a un seul serveur qui gère tous les nœuds du système.

4.3 Implémentation du nœud

Les actions principales du nœud sont : ajout d'un filtre, recherche des filtres correspondants à une requête et suppression d'un filtre dans le système.

4.3.1 Données de chaque nœud

Identifiant du nœud (path) Car le chemin du nœud est unique, donc, nous avons décidé de le considérer comme ID d'un nœud. Ce chemin est une chaîne de caractères.

Niveau du nœud dans le système (rang) est hauteur du nœud dans le système.

Table de routage (localRoute) est une table d'une dimension, elle contient l'adresse soit d'un conteneur local, soit d'un conteneur à distance, soit aussi du vide. L'index d'ajout des données dans cette table est calculé par le fragment du rang de son nœud.

En outre, le nœud doit connaître son serveur qui l'héberge.

4.3.2 Ajout'd'un filtre dans le nœud

L'action d'ajout d'un filtre de Bloom dans un nœud se déroule selon les étapes suivantes :

1. Il détermine la valeur du fragment de rang i du filtre de la requête. Cette valeur est l'index dans la table *localRoute*
2. Ensuite, il y a 3 cas à distinguer :
   (a) Si cette entrée est vide, donc, il crée un conteneur local auquel il ajoute ce filtre.
   (b) Si cette entrée contient l'adresse d'un nœud fils, ce nœud retourne au système cette adresse.
   (c) Si cette entée contient déjà un conteneur local, si ce conteneur a des places disponibles, il l'ajoute. Sinon, il renvoie ce conteneur au système pour que le système puisse créer un nouveau fils correspondant à cette entrée. Le système appelle la méthode **spilt()** et répond au nœud appelant l'adresse de son nouveau fils. Cette adresse est ajoutée dans sa table *localRoute* en remplaçant le conteneur local précédent.

Algorithme 1 *Ajout d'un filtre de Bloom dans le nœud* ***add***

IN : BF *bf*
OUT : Object *o*

---

1. Fragment $f \leftarrow$ *bf*.getFragment(rang)
2. Object $o \leftarrow$ *localRoute*.get(f)
3. **if** $o == null$
   ContainerLocal $c \leftarrow$ new ContainerLocal
   c.add(*bf*)
   localRoute.put(f, c)
   return *null*
   else
   return o
   end

4.3.3 Recherche des filtres dans le nœud

Le nœud reçoit la requête sous forme d'un filtre de Bloom :

1. Comme l'action **add**, il doit déterminer le fragment de rang i de la requête.
2. Il parcourt toute la table *localRoute* pour trouver tous les entrées non vides qui contiennent le fragment déterminé de la requête :
   (a) Si l'entrée correspondante contient l'adresse du conteneur local dont il récupère tous les filtres qui contiennent celui de la requête et ajoute dans l'ensemble de résultats.
   (b) Si l'entrée correspondante contient l'adresse du nœud fils, il l'ajoute également dans l'ensemble de résultats.

Algorithme 2 *Recherche des filtres de Bloom dans le nœud* ***search***

IN : BF *bf*
OUT : Object *o*

---

1. Fragment $f \leftarrow$ *bf*.getFragment(rang)
2. Object $o \leftarrow$ *null*
3. Integer i $\leftarrow$ 0
4. **while** i $\leq$ *localRoute*.size()
   Object *tmp* $\leftarrow$ *null*
   if ($f \in i$) && ((*tmp* $\leftarrow$ *localRoute*.get(i)) $\neq$ *null*)
   if *tmp* **instance of** ContainerLocal
   while *tmp*.hasNext()
   BF *bf_tmp* $\leftarrow$ *tmp*.next()
   if *bf* $\in$ *bf_tmp*
   o.add(*bf_tmp*)
   end
   end
   else /* *tmp* **instance of** String */
   o.add(*tmp*)
   end
   end
   i++
   end
5. **return** *o*

4.3.4 Suppression d'un filtre dans le nœud

Lors de la suppression d'un filtre de Bloom dans le nœud, les étapes sont :

1. D'abord, il détermine la valeur de fragment de rang i du filtre de la requête.
2. Si *localRoute* ne contient pas ce fragment, donc, il n'y a pas ce filtre dans le système.
3. S'il existe un entrée non vide dans la table *localRoute*, alors :
   (a) Si cette entrée contient un conteneur local, alors il supprime le filtre de la requête dans ce conteneur. Si ce conteneur devient vide, il le supprime dans la table *localRoute*. En plus, si cette table devient aussi vide, il renvoie cette table au système pourqu'il puisse supprimer le nœud qui la contient.
   (b) Si cette entrée contient l'adresse d'un nœud fils, il renvoie cette adresse au système.

Algorithme 3 *Suppression d'un filtre de Bloom dans le nœud* ***remove***

IN : BF *bf*
OUT : Object *o*

---

1. Fragment $f \leftarrow$ *bf*.getFragment(rang)
2. **if** !*localRoute*.contains(f)
   return *null*
   end
3. Object $o \leftarrow$ *localRoute*.get(f)
4. **if** *o* **instance of** ContainerLocal
   o.remove(*bf*)
   if *o*.isEmpty()
   localRoute.remove(f)
   end
   if *localRoute*.isEmpty()
   return *localRoute*
   end
   return *null*
   end
5. **return** *localRoute*.get(f) /*retourne l'adresse de nœud fils*/

4.4 Implémentation du système d'index

Le système d'index gère les nœuds. Les actions autorisées pour les utilisateurs sont : ajout d'un filtre de Bloom, recherche des filtres de Bloom et supprime d'un filtre de Bloom. En plus, il existe des actions internes du système comme **split**.

4.4.1 Données du système d'index

Identifiant de l'index (indexID) est pour distinguer entre plusieurs index différents.

Identifiant du server (serverID) est l'identifiant du server qui gère cet index.

Capacité maximale d'un conteneur local (gamma) est la taille maximum du conteneur local du système.

Table de nœuds (listNode) contient tous les nœuds dans le système.

4.4.2 Eclatement d'un niveau dans le système

Cette méthode est appelée lorsque le conteneur local est plein, elle prend le conteneur local et le nœud père contenant ce conteneur en entrée. D'abord, elle crée le nœud qui corresponde au conteneur. Ensuite, pour chaque élément, elle envoie vers ce nouveau nœud en appelant la méthode **add**. A la fin, elle ajoute l'adresse de ce nouveau nœud dans la table *localRoute* de son père.

Algorithme 4 *Eclatement d'un niveau dans le système* ***split***

IN : Node *father*, ContainerLocal *c*
OUT : *null*

---

1. BF *bf* ← *c*.get(0)
2. Fragment *f* ← *bf*.getFragment(*father*.getRang())
3. String *path* ← *father*.getPath() + *"/"* + *f*
4. Integer *newRang* ← *father*.getRang() + 1
5. Node *n* ← new Node(*null*, *path*, *newRang*, *gamma*) /*ici, serverID ← *null*/
6. *father*.add(*bf*, *path*)
7. *listNode*.put(*path*, *n*)
8. Integer *i* ← 0
9. **while** *i* < *c*.size()
   n.add(*c*.get(i))
   i++
   end

4.4.3 Ajout d'un filtre de Bloom dans le système

Lors de la réception d'une demande d'ajout dans le système, il renvoie cette requête vers le nœud racine, c'est-à-dire le nœud de niveau 0. Il y a 3 cas possibles :

1. La réponse du nœud est son conteneur local, le système appelle la méthode **split** et il continue à traiter les autres requêtes.
2. La réponse du nœud est l'identifiant de son fils, le système cherche ce nœud fils et lui renvoie cette requête.
3. La réponse du nœud ne contient aucun élément, c'est le succès.

Le système continue ce procédure jusqu'à quand l'insertion de ce filtre réussit.

Algorithme 5 *Ajout d'un filtre de Bloom dans le système* ***add***

IN : BF *bf*
OUT : *null*

---

1. Node *n* ← *listNode*.get(*"/"*)
2. Object *o* ← *n*.add(*bf*)
3. **while** *o* ≠ *null*
   if *o* **instance of** ContainerLocal
   split(*n*, *o*)
   o ← *null*
   else
   n ← *listNode*.get(*o*)
   o ← *n*.add(*bf*)
   end
   end

4.4.4 Recherche des filtres de Bloom dans le système

Pour la recherche des filtres de Bloom qui matchent la requête, le système renvoie la requête au nœud racine et reçoit le résultat à partir de ce nœud. Le nœud répond au système en envoyant un ensemble de données dans lequel il y a 2 types de données possibles :

1. soit le filtre de Bloom, simplement, le système l'ajoute dans l'ensemble de filtres trouvés.
2. soit l'identifiant du nœud fils, alors, il doit y envoyer la requête et traiter sa réponse.
3. soit rien.

A la fin, il retourne le résultat complet.

Algorithme 6 *Recherche des filtres de Bloom dans le système* ***search***

IN : BF *bf*
OUT : ArrayList<BF> *resultat*

---

1. Node $n \leftarrow$ *listNode*.get(*"/"*)
2. ArrayList<Object> *list* $\leftarrow$ *n*.search(*bf*)
3. ArrayList<BF> *resultat* $\leftarrow$ *null*
4. **while** *list*.hasNext()
   Object *o* $\leftarrow$ *list*.next()
   if *o* **instance of** BF
   resultat.add(*o*)
   else
   Node *tmp* $\leftarrow$ *listNode*.get(*o*)
   list.add(*tmp*.search(*bf*))
   end
   end
5. **return** *resultat*

4.4.5 Suppression d'un filtre de Bloom dans le système

La requête de suppression est aussi envoyée au nœud racine. Il y aussi 3 cas à distinguer :

1. Si la réponse est l'identifiant de son nœud fils, le système y renvoie la requête et attend la réponse.
2. Si la réponse est une table de routage, cela veut dire que cette table est vide après la suppression de son dernier élément. Le système demande son père de supprimer l'entrée correspondante avec le nœud qui gère cette table et supprime ce nœud, si la table de routage de son père devient vide, le système remonte ce procédure vers la racine pour finir la suppression des nœuds vides.
3. Si la réponse est null, cette méthode finit avec succès.

Algorithme 7 *Suppression d'un filtre de Bloom dans le système* ***remove***

IN : BF *bf*
OUT : *null*

---

1. Node $n \leftarrow$ *listNode*.get(*"/"*)
2. Object $o \leftarrow n$.remove(bf)
3. **while** $o \neq null$
   if o **instance of** String /*l'adresse d'un nœud est une chaine de caractères*/
   $n \leftarrow$ *listNode*.get(o)
   $o \leftarrow n$.remove(bf)
   else
   String *path* $\leftarrow n$.getPath() /*retourne l'adresse de ce nœud*/
   Integer *rang* $\leftarrow n$.getRang()
   if *path* == *"/"*
   return
   end
   listNode.remove(*path*)
   Integer *lastIndex* $\leftarrow$ *path*.lastIndexOf(*"/"*) /*retourne la dernière position de cette caractère*/
   $n \leftarrow$ *listNode*.get(*path*.substring(0, *lastIndex*))
   while *true*
   if n.remove(*bf*.getFragment(*rang*))
   return
   else
   path $\leftarrow n$.getPath()
   rang $\leftarrow n$.getRang()
   if *path* == *"/"*
   return
   end
   listNode.remove(*path*)
   lastIndex $\leftarrow$ *path*.lastIndexOf(*"/"*)
   $n \leftarrow$ *listNode*.get(*path*.substring(0, *lsatIndex*))
   end
   end
   end
   end