8INF840

Projet final : skiplists



BRANGER Mathias

CROUZET Matthieu

## 

[**Introduction**](#_2q39d1k2lc6z)[**2**](#_2q39d1k2lc6z)

[**Définition des Skiplist**](#_r5klw3plpgyz)[**2**](#_r5klw3plpgyz)

[Recherche](#_kradtoa4vnhn) [3](#_kradtoa4vnhn)

[Insertion](#_svlz6hhqla5x) [4](#_svlz6hhqla5x)

[Suppression](#_wbibuvulf5pj) [5](#_wbibuvulf5pj)

[**Domaines d’application**](#_ysn80eyqcgad)[**6**](#_ysn80eyqcgad)

[**Implémentation**](#_ry10bplpw5jr)[**6**](#_ry10bplpw5jr)

[A- Introduction](#_immn3m555zom) [6](#_immn3m555zom)

[B- Définition de la structure d’une skiplist](#_yoafhsca4dkl) [7](#_yoafhsca4dkl)

[C- Algorithmes utilisés](#_3zfsln5gdy8z) [8](#_3zfsln5gdy8z)

[**Résultats**](#_kh0159mwh0eo)[**8**](#_kh0159mwh0eo)

[**Conclusion**](#_7xsiglqrd7q1)[**9**](#_7xsiglqrd7q1)

## 

## 

## Introduction

Chaque problème peut être modélisé de plusieurs manières. En choisissant de le modéliser d’une certaine méthode, il peut être possible de le résoudre très efficacement. Dans l’informatique, il est alors intéressant d’étudier les différentes structures de données pour savoir à quel problème on peut l’appliquer.

Dans ce document, nous étudierons la structure de données appelée skiplist.

## Définition des Skiplist

Pour définir les skipslists nous allons les construire pas à pas.

Les skiplists sont une amélioration les listes chaînées triées. Il est donc d’abord important de rappeler les différentes complexités liées à cette structure :

|  |  |
| --- | --- |
| Algorithme | Complexité |
| Insertion | O(n) |
| Recherche | O(n) |
| Suppression | O(n) |

Une skiplist est un ensemble de liste chaînées triées structurées en couches. Dans l’exemple suivant, on peut voir une skiplist avec 2 couches.



Si je suis à l’élément 12 et que je souhaite accéder à l’élément 52, je vais emprunter le chemin suivant :



Les skiplists semblent alors être efficaces pour rechercher des éléments.

Il apparaît donc qu’elles doivent respecter les **propriétés** suivantes :

* C’est un ensemble de listes chaînées triées en couches.
* La couche doit 0 contenir tous les noeuds.
* La couche i -1 doit contenir au moins toutes les noeuds de la couche i.
* Chaque couche contient au moins le premier et le dernier élément de la couche 0

### Recherche

Comme nous l’avons vu dans l’exemple précédent, l’algorithme consiste à partir du premier élément de la couche la plus haute, de parcourir les éléments de la couche, et de réitérer sur la couche inférieure si l’élément suivant est trop grand par rapport à l’élément recherché. Si on ne trouve pas l’élément dans la dernière couche alors il n’est pas présent.

Autrement dit, le pseudo-code correspondant est le suivant :



Le nombre d’éléments étant moins élevé dans les couches supérieures, il est évident que la recherche est plus rapide que dans une simple liste chaînée. Mais à quel point ?

Si on reprend l’exemple précédent, pour une recherche dans une skiplist avec 2 couches, on va d’abord parcourir la couche 1 puis il y a la possibilité de parcourir une partie de la couche 0. On obtient donc un coût d'environ **|C1| + ( |C0| / |C1| ).**

Si on veut minimiser ce coût avec n éléments, il faut que ces éléments soient espacés tels que : **|C1|² = |C0| = n** et donc **|C1| = √n**

Les éléments de la couche 1 doivent donc être répartis uniformément avec un espace de taille **√n** qui nous permettra d’obtenir une complexité en O(**√n**)



Si on recommence le raisonnement avec un nombre différent de couche, on va obtenir les complexités suivantes :

|  |  |
| --- | --- |
| Nombre de couches | Complexité de recherche |
| 3 |  |
| k |  |
| log n |  |

La meilleur complexité que l’on puisse obtenir est donc O(log n) et elle s’obtient avec log n couches.

Il existe d’autres algorithmes de recherche. L’un va appliquer le même algorithme que précédemment sauf qu’il part de la queue de la liste. L’autre va choisir de descendre d’une couche plus tard pour ensuite revenir en arrière.

Un problème se pose alors : comment créer une telle structure ?

### Insertion

Un algorithme naïf d’insertion dans une skiplist consisterait à rechercher l’emplacement où l’élément devrait être situé, et de le lier aux éléments précédent et suivant.

Mais selon l'algorithme de recherche, on se repère par rapport à l’élément suivant pour savoir si on doit insérer le nouvel élément à l’emplacement courant. Or, si on veut ajouter un élément en tête de liste, c’est impossible.

Pour pallier à ce problème, la tête de chaque couche doit commencer par un élément fictif avec une clé -∞. Ainsi, on peut obtenir le cas suivant :



A ce stade, l’insertion n’est toujours pas correcte puisqu’elle n’assure pas une répartition uniforme décroissante vers le haut à travers les couches.

La répartition uniforme est faite par l’algorithme probabiliste suivant :



On insère l’élément dans la couche 0 grâce à l’algorithme précédent. Ensuite, on va “tirer à pile ou face” si l’élément va être inséré dans la couche supérieure. Tant qu’on a “pile”, on insère l’élément dans la couche supérieure. Dès qu’on a “face”, on s’arrête.

De cette façon, au plus on aura d’éléments dans notre structure, au plus notre structure tendra vers (log n) couches uniformément répartis. Autrement dit, la complexité de la recherche sera en O(log n).

La skiplist est donc une structure de données probabiliste.

L’insertion étant basée sur la recherche, on obtient la même complexité en O(log n).

### Suppression

La suppression d’un élément n’a rien de particulier, il suffit d’appliquer l’algorithme suivant:



La suppression étant basée sur la recherche, on obtient aussi une complexité de suppression en O(log n).

## Domaines d’application

Cette structure de données n’est pas très utilisée même si on peut la retrouver dans des SGBD comme memSQL, NessDB, SkipDB ou depuis l’API java 1.6.

Pour notre projet, nous avons implémenté les skiplists pour résoudre un problème de transport. En effet, on peut représenter une ligne de transport à l’aide des skiplists. Dans l’exemple ci-dessous, on peut voir une ligne de transport qui dessert des gares entre Lyon et Grenoble. Chaque couleur représente une couche différente. Sur cette ligne, il y a une expresse, ainsi qu’une expresse de l’expresse. On a donc une skiplist à 3 couches.



On peut par contre se demander si cette ligne est uniformément répartie. Dans l’exemple, il y a 17 arrêts, or ln(17) = 2.8 soit environ 3. On retrouve bien 3 lignes.

Il y a 17 arrêts, or √(17) ~ 4 et ∛(17) ~ 2,6. On retrouve bien en moyenne 4 arrêts entre les gares rouges et 2 entre les gares bleues.

La ligne respecte donc les propriétés de la skiplist, ce qui parait normal puisque le but est de desservir un maximum d’usagers et de les déplacer d’un point à un autre le plus rapidement possible. On s’attend à ce que les réseaux de transports avec des lignes expresses respectent également ce modèle.

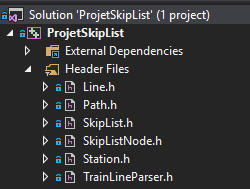
## Implémentation

### A- Introduction

Nous avons donc choisi d’implémenter des skiplists pour modéliser un problème de transport du type : “**Je suis à l’arrêt X et je veux me rendre à l’arrêt Y. Quel est le chemin le plus rapide ?**”.

Pour répondre à un tel problème, nous allons utiliser des lignes de transports déjà existantes. L’insertion des éléments dans notre structure ne se fera donc pas de manière probabiliste. Heureusement, on a vu précédemment que les lignes de transports respectent l’uniformité définie par les skiplist.

Voici les différentes classes que nous avons implémentés :



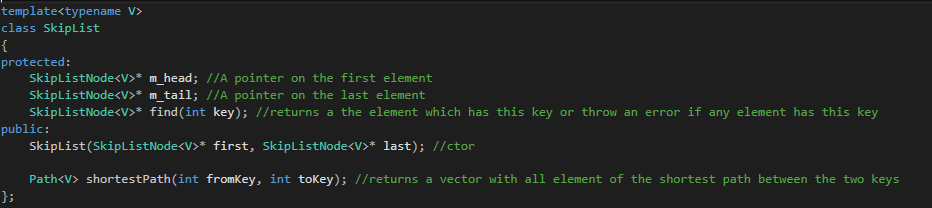
En résumé :

* Une SkipList est donc définie par un ensemble de SkipListNode.
* Une Station contient le nom de la gare.
* Une Line est une SkipList appliquée à un ensemble de Station. Il est possible de parser un fichier à l’aide de TrainLineParser pour parser une Line.
* Un Path défini un chemin dans une SkipList.

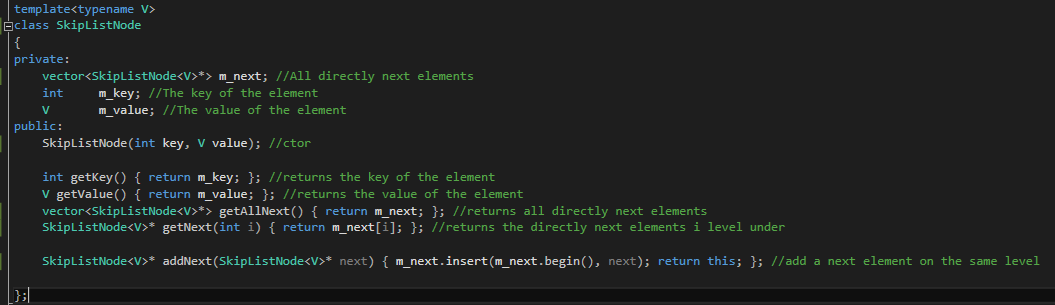
### 

### B- Définition de la structure d’une skiplist

Nous avons choisi de définir **une SkipList** de la façon suivant :



Et **chaque noeud** est défini de la façon suivante :



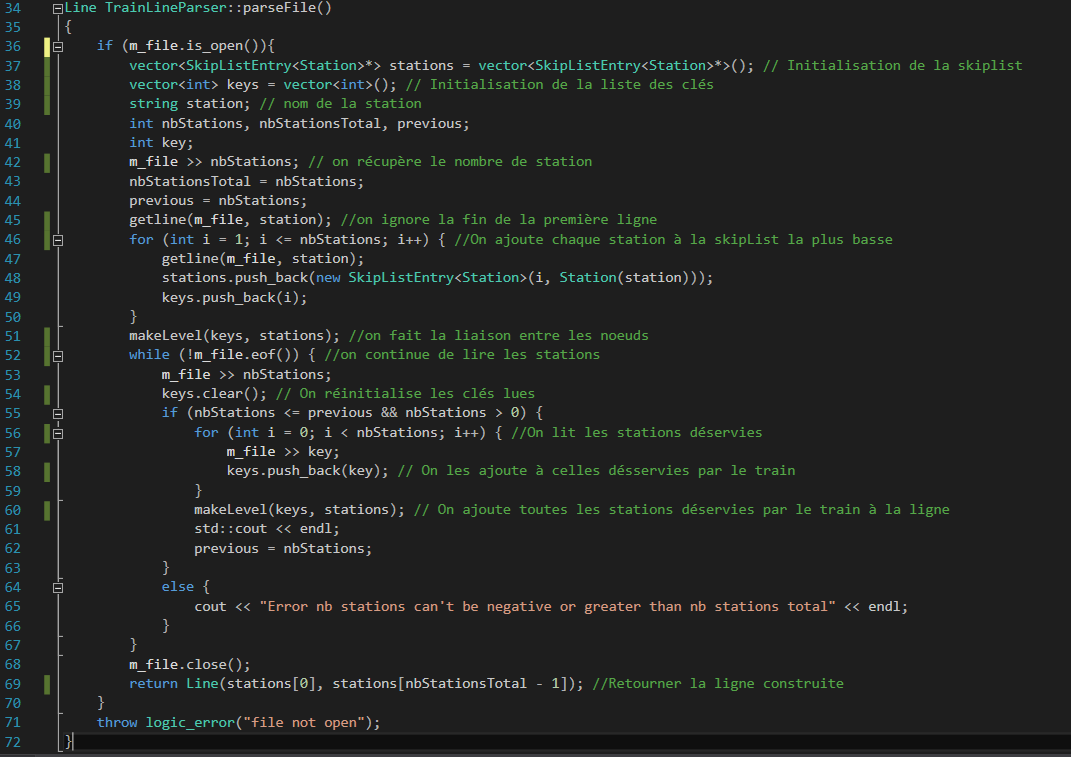
Ainsi, nous avons avons décidé de regrouper les noeuds présents sur plusieurs couches en un seul noeud. Notre système ressemble donc au schéma suivant :



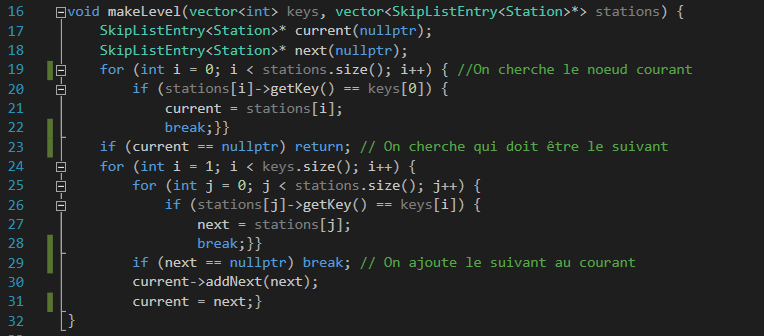
Dans ce cas, pour accéder au noeud 17 en utilisant notre structure, il faudrait accéder à la tête de la SkipList et appeler la fonction getNext(0). Pour le noeuf 15 on fait de même mais en appelant la fonction getNext(1).

### C- Algorithmes utilisés

Voici les algorithmes que nous avons utilisé pour faire fonctionner cette structure de données.

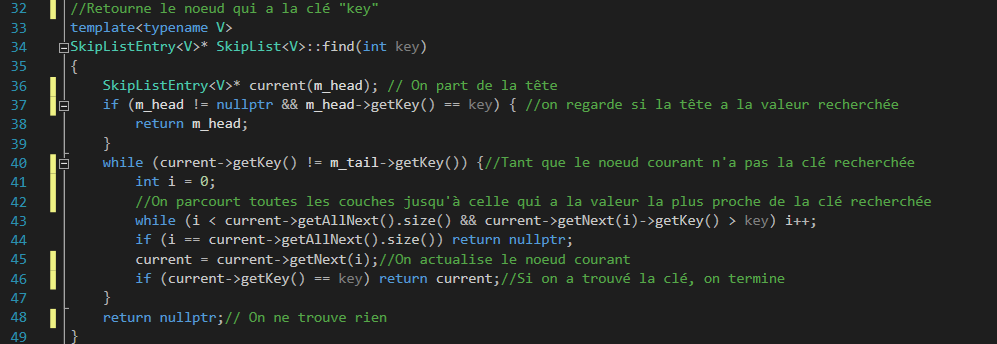
* Insertion

Dans laquelle on utilise la fonction suivante :

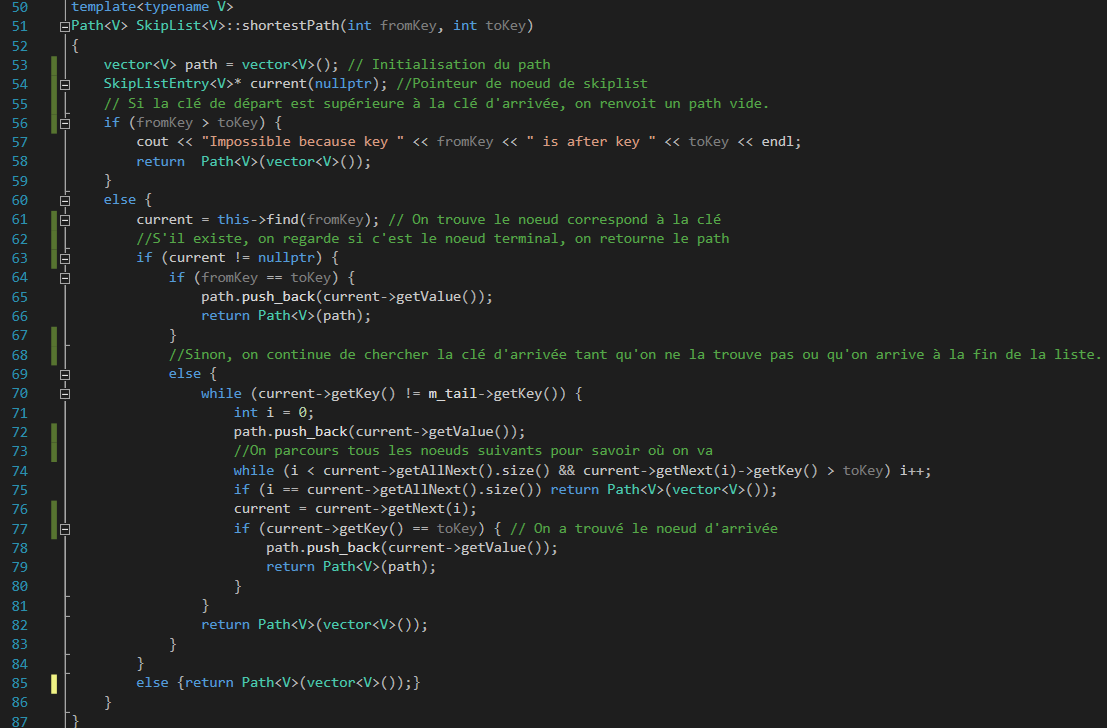


On notera ici que nous n’avons pas implémenté de solution probabiliste puisqu’on veut nos stations à des emplacements précis.

* Recherche



* Shortest path

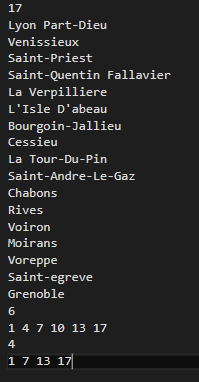


On notera que nous n’avons pas implémenté la suppression d’un noeud car nous n’en n’avons pas besoin. L’algorithme consiste à trouver le noeud à supprimer, lier son prédécesseur à son successeur sur toutes les couches. Il faudrait donc d’abord qu’on implémente le double lien entre les noeuds.

## 

## Résultats

Grâce à notre implémentation, il est possible de parser le fichier suivant (correspondant à notre exemple) :



nombre de gare = N

gare1

gare2

…

gareN

nombre de gare sur l’expresse1 = N1 <= N; N1 >= 2

indice des gares dans l’ordre (doit contenir 1 et N)

nombre de gare sur l’expresse2 = N2 <= N1; N2 >= 2

indice des gares dans l’ordre (contient uniquement des gares de l’expresse1 dont 1 et N)

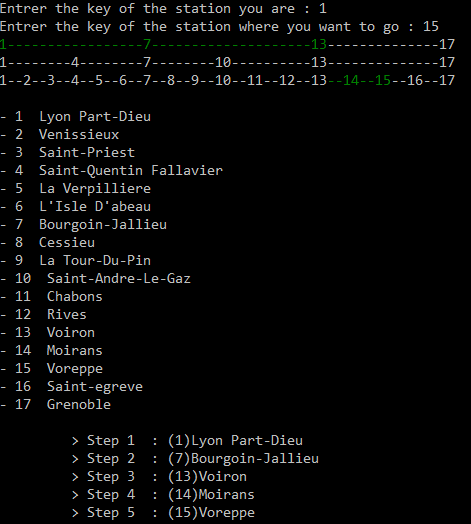
…

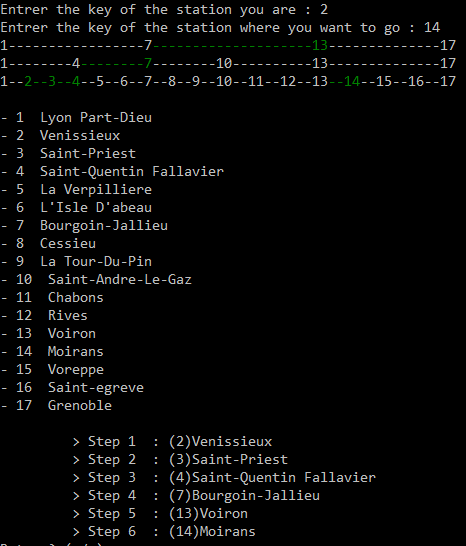
nombre de gare sur l’expressek = Nk <= Nk-1; N2 >= 2

indice des gares dans l’ordre (contient uniquement des gares de l’expressek-1 dont 1 et N)

Et de demander un plus court chemin d’un point à un autre.

Voici quelques exemples :





## 

## Conclusion

Si on résume ce que l’on a vu jusqu’ici, une skiplist est une structure de données probabiliste qui respecte les complexités suivantes :

|  |  |
| --- | --- |
| Algorithme | Complexité |
| Insertion | O(log n) |
| Recherche | O(log n) |
| Suppression | O(log n) |

Ces complexités peuvent s’apparenter à un arbre binaire équilibré mais la skiplist a l’avantage d’être moins bloquante lors d’accès concurrents.

Notre implémentation permettant de modéliser une ligne de transport et de demander un plus court chemin entre deux points fonctionne correctement.