26/11/2016

8INF840

Projet final : Skiplists

BRANGER Mathias
CROUZET Matthieu

UQAC
Université du Québec à Chicoutimi

Table des matières

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.

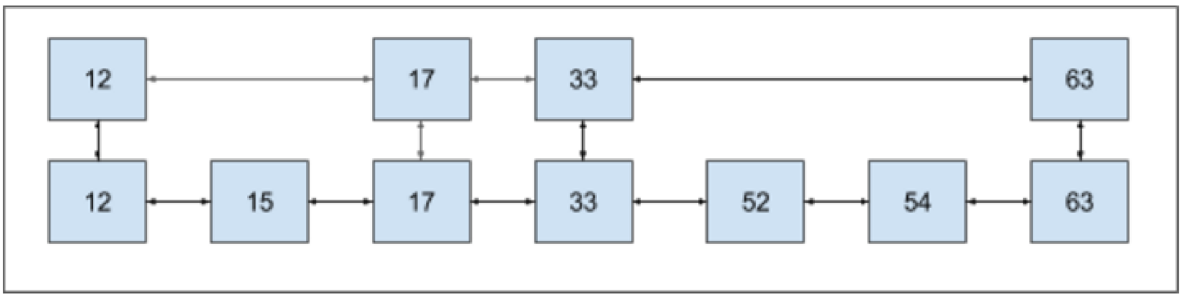
1. Définition des Skiplist

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

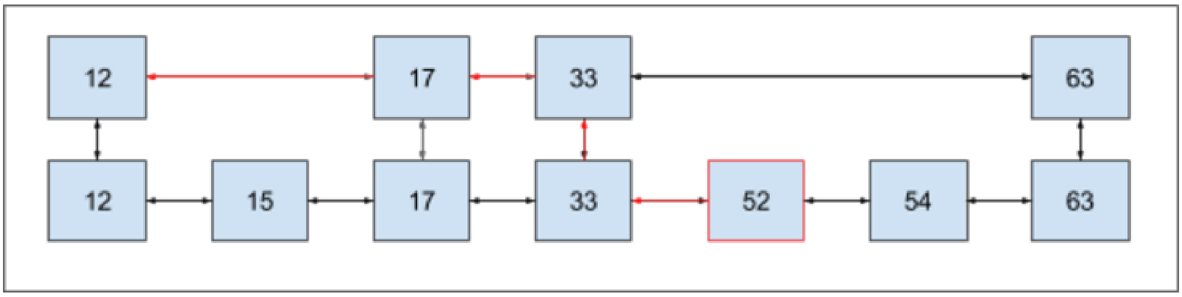
Les skiplists sont une amélioration les listes chaînées triées. Il est donc d'abord important de rappeler les différentes complexités liées à cette structure :

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

a. 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 :

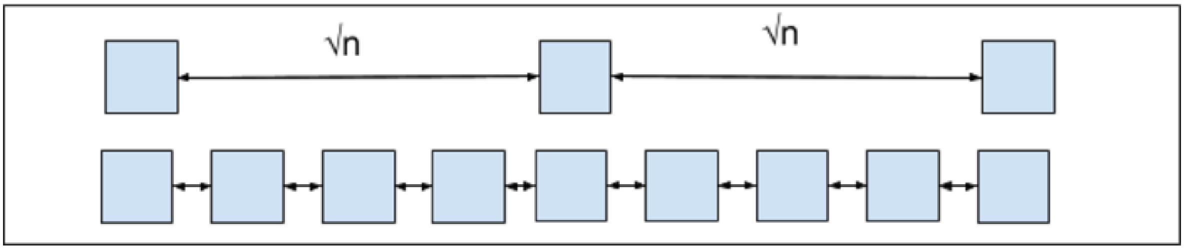
```
Recherche (X):
        Recherche_rec(couche[max]->tête, X)

Recherche_rec (élément, X) :
        Tant que élément⟶suivant⟶clé < X
                Recherche_rec(élément⟶suivant, X)

        si élément⟶clé = X
                retourner élément
        Si élément⟶clé > X ET élément⟶couche > 0
                Recherche_rec(élément⟶coucheInferieur⟶suivant, X)
        Sinon
                retourner non trouvé
```

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	$O(\sqrt[3]{n})$
k	$O(\sqrt[k]{k})$
log n	$\sqrt[\log n]{\log n} = O(\log n)$

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

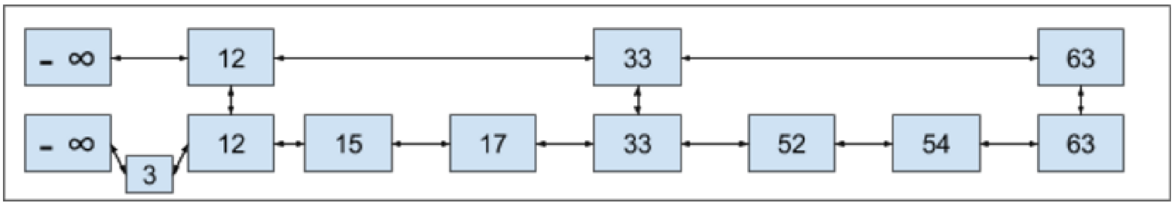
Il existe d'autres algorithmes de recherche. L'un va appliquer le même algorithme que précédemment sauf qu'il part de la queue de la liste. L'autre va choisir de descendre d'une couche plus tard pour ensuite revenir en arrière.
Un problème se pose alors : comment créer une telle structure ?

b. 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 précédant 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 :

```
Insertion (X) :
    Rechercher l'emplacement de X comme pour une simple recherche
    Insérer X dans L0 et changer les liens
    P = aléatoire (pile, face)
    couche= 0

    Tant que P == pile
        Si couche> nombre de niveaux
            Créer nouvelle couche et y insérer X
            sortir du while
        Insérer X dans C[couche] et changer les liens
        P = aléatoire (pile, face)
        couche++
```

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).

c. Suppression

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

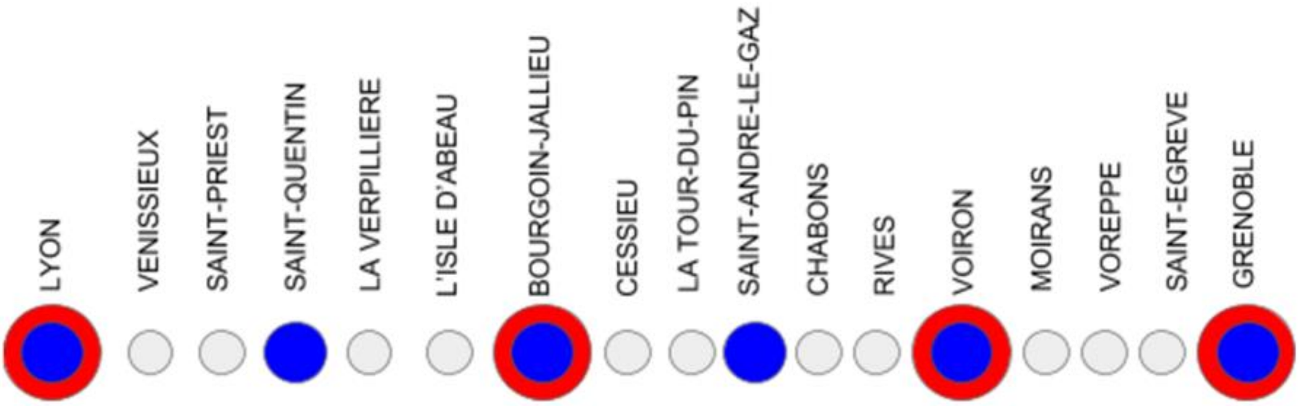
```
Suppression (X):
    Rechercher (X)
    Pour chaque niveau, lier les prédécesseurs de X aux successeurs de X
    Supprimer X de tous les niveaux
```

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

2. 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 $\sqrt{17} \sim 4$ et $\sqrt[3]{17} \sim 2{,}6$. 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.

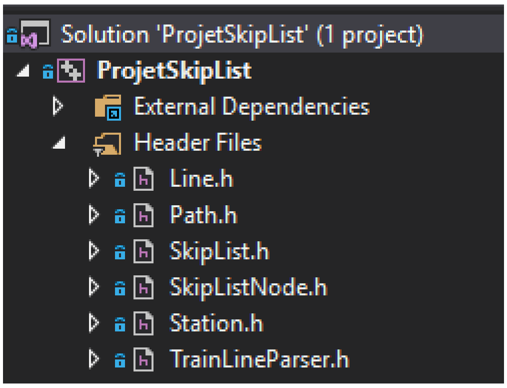
3. Implémentation

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 :

```
template<typename V>
class SkipList
{
protected:
    SkipListNode<V>* m_head; //A pointer on the first element
    SkipListNode<V>* m_tail; //A pointer on the last element
    SkipListNode<V>* find(int key); //returns a the element which has this key or throw an error if any element has this key
public:
    SkipList(SkipListNode<V>* first, SkipListNode<V>* last); //ctor

    Path<V> shortestPath(int fromKey, int toKey); //returns a vector with all element of the shortest path between the two keys
};
```

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

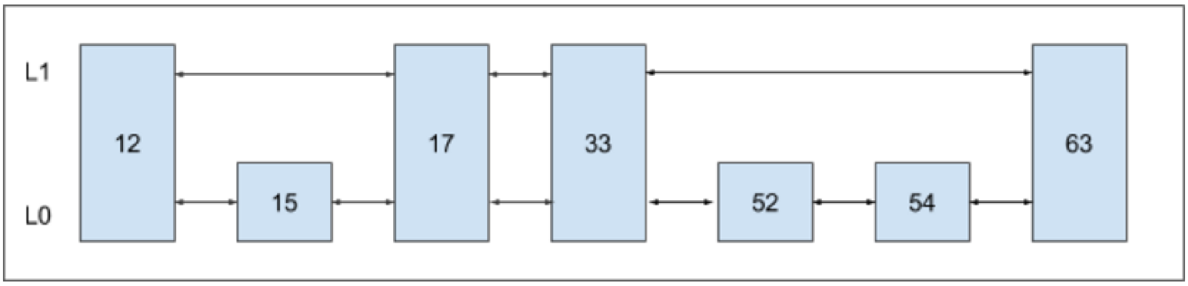
```
template<typename V>
class SkipListNode
{
private:
    vector<SkipListNode<V>*> m_next; //All directly next elements
    int     m_key; //The key of the element
    V       m_value; //The value of the element
public:
    SkipListNode(int key, V value); //ctor

    int getKey() { return m_key; }; //returns the key of the element
    V getValue() { return m_value; }; //returns the value of the element
    vector<SkipListNode<V>*> getAllNext() { return m_next; }; //returns all directly next elements
    SkipListNode<V>* getNext(int i) { return m_next[i]; }; //returns the directly next elements i level under

    SkipListNode<V>* addNext(SkipListNode<V>* next) { m_next.insert(m_next.begin(), next); return this; }; //add a next element on the same level

};
```

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

```cpp
Line TrainLineParser::parseFile()
{
    if (m_file.is_open()){
        vector<SkipListEntry<Station>*> stations = vector<SkipListEntry<Station>*>(); // Initialisation de la skiplist
        vector<int> keys = vector<int>(); // Initialisation de la liste des clés
        string station; // nom de la station
        int nbStations, nbStationsTotal, previous;
        int key;
        m_file >> nbStations; // on récupère le nombre de station
        nbStationsTotal = nbStations;
        previous = nbStations;
        getline(m_file, station); //on ignore la fin de la première ligne
        for (int i = 1; i <= nbStations; i++) { //On ajoute chaque station à la skipList la plus basse
            getline(m_file, station);
            stations.push_back(new SkipListEntry<Station>(i, Station(station)));
            keys.push_back(i);
        }
        makeLevel(keys, stations); //on fait la liaison entre les noeuds
        while (!m_file.eof()) { //on continue de lire les stations
            m_file >> nbStations;
            keys.clear(); // On réinitialise les clés lues
            if (nbStations <= previous && nbStations > 0) {
                for (int i = 0; i < nbStations; i++) { //On lit les stations déservies
                    m_file >> key;
                    keys.push_back(key); // On les ajoute à celles désservies par le train
                }
                makeLevel(keys, stations); // On ajoute toutes les stations déservies par le train à la ligne
                std::cout << endl;
                previous = nbStations;
            }
            else {
                cout << "Error nb stations can't be negative or greater than nb stations total" << endl;
            }
        }
        m_file.close();
        return Line(stations[0], stations[nbStationsTotal - 1]); //Retourner la ligne construite
    }
    throw logic_error("file not open");
}
```

Dans laquelle on utilise la fonction suivante :

```cpp
void makeLevel(vector<int> keys, vector<SkipListEntry<Station>*> stations) {
    SkipListEntry<Station>* current(nullptr);
    SkipListEntry<Station>* next(nullptr);
    for (int i = 0; i < stations.size(); i++) { //On cherche le noeud courant
        if (stations[i]->getKey() == keys[0]) {
            current = stations[i];
            break;}}
    if (current == nullptr) return; // On cherche qui doit être le suivant
    for (int i = 1; i < keys.size(); i++) {
        for (int j = 0; j < stations.size(); j++) {
            if (stations[j]->getKey() == keys[i]) {
                next = stations[j];
                break;}}
        if (next == nullptr) break; // On ajoute le suivant au courant
        current->addNext(next);
        current = next;}
}
```

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

➔ Recherche

```cpp
//Retourne le noeud qui a la clé "key"
template<typename V>
SkipListEntry<V>* SkipList<V>::find(int key)
{
    SkipListEntry<V>* current(m_head); // On part de la tête
    if (m_head != nullptr && m_head->getKey() == key) { //on regarde si la tête a la valeur recherchée
        return m_head;
    }
    while (current->getKey() != m_tail->getKey()) {//Tant que le noeud courant n'a pas la clé recherchée
        int i = 0;
        //On parcourt toutes les couches jusqu'à celle qui a la valeur la plus proche de la clé recherchée
        while (i < current->getAllNext().size() && current->getNext(i)->getKey() > key) i++;
        if (i == current->getAllNext().size()) return nullptr;
        current = current->getNext(i);//On actualise le noeud courant
        if (current->getKey() == key) return current;//Si on a trouvé la clé, on termine
    }
    return nullptr;// On ne trouve rien
}
```

➔ Shortest path

```cpp
template<typename V>
Path<V> SkipList<V>::shortestPath(int fromKey, int toKey)
{
    vector<V> path = vector<V>(); // Initialisation du path
    SkipListEntry<V>* current(nullptr); //Pointeur de noeud de skiplist
    // Si la clé de départ est supérieure à la clé d'arrivée, on renvoit un path vide.
    if (fromKey > toKey) {
        cout << "Impossible because key " << fromKey << " is after key " << toKey << endl;
        return  Path<V>(vector<V>());
    }
    else {
        current = this->find(fromKey); // On trouve le noeud correspond à la clé
        //S'il existe, on regarde si c'est le noeud terminal, on retourne le path
        if (current != nullptr) {
            if (fromKey == toKey) {
                path.push_back(current->getValue());
                return Path<V>(path);
            }
            //Sinon, on continue de chercher la clé d'arrivée tant qu'on ne la trouve pas ou qu'on arrive à la fin de la liste.
            else {
                while (current->getKey() != m_tail->getKey()) {
                    int i = 0;
                    path.push_back(current->getValue());
                    //On parcours tous les noeuds suivants pour savoir où on va
                    while (i < current->getAllNext().size() && current->getNext(i)->getKey() > toKey) i++;
                    if (i == current->getAllNext().size()) return Path<V>(vector<V>());
                    current = current->getNext(i);
                    if (current->getKey() == toKey) { // On a trouvé le noeud d'arrivée
                        path.push_back(current->getValue());
                        return Path<V>(path);
                    }
                }
                return Path<V>(vector<V>());
            }
        }
        else {return Path<V>(vector<V>());}
    }
}
```

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.

4. Résultats

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

```
17
Lyon Part-Dieu
Venissieux
Saint-Priest
Saint-Quentin Fallavier
La Verpilliere
L'Isle D'abeau
Bourgoin-Jallieu
Cessieu
La Tour-Du-Pin
Saint-Andre-Le-Gaz
Chabons
Rives
Voiron
Moirans
Voreppe
Saint-egreve
Grenoble
6
1 4 7 10 13 17
4
1 7 13 17
```

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 :

```
Entrer the key of the station you are : 1
Entrer the key of the station where you want to go : 15
1-----------------7--------------------13--------------17
1--------4--------7--------10----------13--------------17
1--2--3--4--5--6--7--8--9--10--11--12--13--14--15--16--17

- 1  Lyon Part-Dieu
- 2  Venissieux
- 3  Saint-Priest
- 4  Saint-Quentin Fallavier
- 5  La Verpilliere
- 6  L'Isle D'abeau
- 7  Bourgoin-Jallieu
- 8  Cessieu
- 9  La Tour-Du-Pin
- 10  Saint-Andre-Le-Gaz
- 11  Chabons
- 12  Rives
- 13  Voiron
- 14  Moirans
- 15  Voreppe
- 16  Saint-egreve
- 17  Grenoble

         > Step 1  : (1)Lyon Part-Dieu
         > Step 2  : (7)Bourgoin-Jallieu
         > Step 3  : (13)Voiron
         > Step 4  : (14)Moirans
         > Step 5  : (15)Voreppe
```

```
Entrer the key of the station you are : 2
Entrer the key of the station where you want to go : 14
1-----------------7--------------------13--------------17
1--------4--------7--------10----------13--------------17
1--2--3--4--5--6--7--8--9--10--11--12--13--14--15--16--17

- 1  Lyon Part-Dieu
- 2  Venissieux
- 3  Saint-Priest
- 4  Saint-Quentin Fallavier
- 5  La Verpilliere
- 6  L'Isle D'abeau
- 7  Bourgoin-Jallieu
- 8  Cessieu
- 9  La Tour-Du-Pin
- 10  Saint-Andre-Le-Gaz
- 11  Chabons
- 12  Rives
- 13  Voiron
- 14  Moirans
- 15  Voreppe
- 16  Saint-egreve
- 17  Grenoble

         > Step 1  : (2)Venissieux
         > Step 2  : (3)Saint-Priest
         > Step 3  : (4)Saint-Quentin Fallavier
         > Step 4  : (7)Bourgoin-Jallieu
         > Step 5  : (13)Voiron
         > Step 6  : (14)Moirans
```

5. 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.