Université de Mons

Master en Sciences informatiques

Skip list

Rapport de lecture et rédaction scientifique

Présenté par Youness KAZZOUL

Directeur : Gwenaël JORET

Juin 2022

Remerciement

Je voudrais tout d'abord adresser toute ma reconnaissance à madame Véronique BRUYERE de l'université de Mons, je la remercie de m'avoir donné l'occasion extraordinaire de réaliser ce travail de rédaction scientifique.

Je souhaiterais remercier mon directeur M. Gwenaël JORET sans oublier le directeur de l'année dernière M. Olivier DELGRANGE de l'université de Mons, leur patience, leur disponibilité et surtout leurs judicieux conseils ont contribué à alimenter ma réflexion.

Je tiens également à témoigner toute ma gratitude à monsieur Vincent CALLUT de l'université de Mons pour son coaching et son aide.

Et enfin, je voudrais aussi adresser mes remerciements à toutes les personnes qui ont contribué au succès de ce travail notamment pour avoir relu et corrigé mon travail ainsi qu'à mes très chers parents, mes sœurs, mes frères et mes amis, pour leurs soutiens constants et leurs encouragements.

À tous ces intervenants, je présente mes remerciements, mon respect et ma gratitude.

Table des matières

1 Introduction

Le monde de l'informatique nécessite une bonne gestion des données pour pouvoir les traiter, les rechercher, les stocker et les supprimer. Aujourd'hui, nous sommes confronter à une lourde quantité de données, qui pour l'essentiel est générée par les outils technologiques de plus en plus performants et de plus en plus importants.

Il est impératif d'avoir une bonne structure de données, selon l'usage qu'il en sera fait, en veillant à réduire la complexité algorithmique pour être rapide et efficace. Il existe différents types de structures de données. Tous les types visent à répondre aux exigences logicielles. *e.g.*, le traitement des données, comme les dictionnaires de données, les services d'indexation, les informations stockées dans des bases de données *etc*, peut être représenté par les ABR (Arbre binaire de recherche), les AVL (autre famille d'arbres binaire de recherche qui sont automatiquement équilibré inventé par Adelson-Velsky et Landis, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. La recherche, l'insertion et la suppression sont toutes en $\mathcal{O}(\log(n)$ dans le pire des cas. L'insertion et la suppression nécessitent d'effectuer des rotations un peut difficile à implementer.) [1], les TAS (heap en anglais, une autre famille d'arbres binaire qui permet de retrouver directement l'élément que l'on veut traiter en priorité, est un arbres binaire presque complet ordonné. *i.e.*, tous ses niveaux sont remplis, sauf éventuellement le dernier, qui doit être rempli sur la gauche. Ses feuilles sont donc à la même distance minimale de la racine, plus ou moins 1. Ont été introduits par J. W. J. Williams en 1964 pour l'algorithme du tri par tas *etc* [2]),... et beaucoup d'autres familles d'arbres binaire.

Les Skip lists ont été présentées en 1989 par William Pugh, un professeur d'informatique à l'Université du Maryland sous l'article intitulé : **Skip lists : A probabilistic alternative to balanced trees** [6] dans lequel il analyse en détail les performances d'une structure de données probabiliste nommée Skip list. Cette structure de données probabiliste à base de listes chaînées parallèles dont les éléments ont une hauteur qui leur est associée, peuvent être une alternative aux arbres équilibrés.

Dans ce travail, nous tenterons d'étudier les Skip lists. La section 2, portera sur les structures de données telles que les listes chaînées ordonnées, les arbres binaires de recherche et les tables de hachage. Dans la section 3, nous analyserons de plus près les Skip lists et je vous présenterai un exemple pour mieux les comprendre. À la section 4, on examinera les algorithmes de recherche, d'insertion et de suppression au travers d'exemples commentés. Nous les analyserons plus en détail à la section 5 pour bien comprendre leur comportement et surtout la complexité temporelle.

2 Structures de données

Cette section portera sur les structures de données telles que les listes chaînées ordonnées qui sont le point de départ des Skip lists. Nous étudierons aussi deux autres structures de données : les arbres binaires de recherche et les tables de hachage.

2.1 Les listes chaînées ordonnées

Pour bien comprendre les Skip lists, dans un premier temps allons voir les listes chaînées ordonnées ou listes simplement liées, car elles sont basées sur ces structures de données.

Définition

Une liste chaînée (en anglais linked list) désigne en informatique une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments de même type, dont la représentation

1. https://fr.wikipedia.org/wiki/Arbre_AVL
2. https://fr.wikipedia.org/wiki/Tas_(informatique)

en mémoire de l'ordinateur est une succession de cellules faites d'un contenu et d'un pointeur vers une autre cellule. De façon imagée, l'ensemble des cellules ressemble à une chaîne dont les maillons seraient les cellules. La Figure 1 montre une liste chaînée ordonnée avec un pointeur vers le début. L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau) [3].

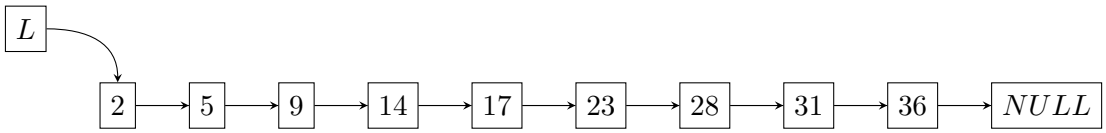

FIGURE 1 – Liste chaînée ordonnée.

Source : *Code repéré en avril 2022 sur https://tex.stackexchange.com, origine de Gonzalo Medina, et le code source développé figure en annexe.*

Rechercher un élément dans une liste chaînée ordonnée

Si on cherche *e.g.*, la derniere clé 36 dans cette liste chaînée ordonnée (L) de la Figure 1, avec un pointeur vers le début (2).

Question : En combien de temps allons nous trouver cette valeur ?
Solution : On doit parcourir la liste en commençant par le début, et on compare notre clé avec chaque clé de la liste, et comme l'élément se trouve à la fin de la liste chaînée, alors il va falloir parcourir toute la liste chaînée.
C'est à dire que dans le pire des cas pour les n éléments, le temps de calcul est de l'ordre de n ou en $\mathcal{O}$(n).

Avantages

- Elles sont des alternatives aux tableaux ;
- Ont une taille dynamique, contrairement aux tableaux, *i.e.*, pour ajouter un nouveau élément à la fin d'un tableau plein, il faut recréer un nouveau tableau plus grand et recopier toutes les ancienes valeurs dans le nouveau tableau, et puis seulement on insère à la fin, tandis que les listes sont dynamiques ;
- Allocation de la mémoire seulement au cas de besoin.

Inconvénients

- Elles ont besoin de plus de mémoire pour stocker les pointeurs ;
- Pour chercher une clé, il faut parcourir tous les nœuds jusqu'à l'élement souhaité ;
- On ne peut pas faire de la recherche dichotomique, contrairement aux tableaux ;
- Le fait qu'une liste soit triée ne nous aide pas à trouver une clé rapidement.

Complexité en temps

Il y a souvent deux buts contradictoires lorsque l'on cherche à mettre au point un algorithme pour résoudre un problème donné :

3. https://fr.wikipedia.org/wiki/Liste_chaînée

1. L'algorithme doit être facile à comprendre, coder, maintenir, mais aussi facile à vérifier.
2. L'algorithme doit utiliser efficacement les ressources de l'ordinateur, c'est-à-dire s'exécuter rapidement, mais aussi prendre une place raisonnable en mémoire.

Si un algorithme doit être utilisé très souvent, il est alors intéressant de mettre en œuvre une solution efficace en temps et/ou en espace mémoire. Il est alors utile de pouvoir comparer objectivement les complexités relatives [5].

Dans cette section et les sections suivantes réservées à la complexité, nous allons se concentrer uniquement sur la complexité en temps.

Rechercher une donnée d dans une liste triée
La recherche d'un élément d dans une liste triée n'est pas compliquer, i.e., on parcourt la liste et si on atteint un élément plus grand que d, on conclu que ça ne sert a rien de continuer a chercher, car tous les éléments suivants sont plus grand que d, malgré cela la complexité est en $\mathcal{O}$(n) au pire des cas. ou n est le nombre des d'éléments dans a liste [3].

Insertion dans une liste triée
Cette opération d'insertion a en entrée une liste triée L et un élément d, le but est d'insérer cet élément d de cette liste, et avoir en sortie une liste triée L'. Pour ce faire, on repère le premier élément. Si il est plus grand que l'élément d, où il n'existe pas *i.e.,* (cas d'une liste est vide), alors on insère notre élément d au début, la complexité est en $\mathcal{O}(1)$, c'est ce qu'on appelle le cas d'insertion au début. Sinon on parcourt la liste jusqu'à trouver un élément k plus petit que d, à ce moment là on insère l'élément d juste après, c'est le cas d'insertion en k^e position, qu'est en $\mathcal{O}$(k).
Il se peut que tous les éléments parcouru sont plus petit que d, à ce moment là on insère à la fin, c'est le pire des cas, la complexité est en $\mathcal{O}$(n) [3].

La suppression dans une liste triée
Cette opération de suppression a en entrée une liste triée L et un élément en position k, le but est de supprimer cet élément, et en sortie avoir a une liste triée L' sans cet élément. Le seul élément qui va être modifier c'est celui qui précède l'élément à supprimer.
La complexité de cette opération est en O(k), car la complexité de recherche est dominante ici [3].

2.2 Les arbres binaires de recherche

Définition Un Arbre Binaire de Recherche (ABR) est une structure de données qui permet d'obtenir rapidement grâce à l'opération de Recherche tous les éléments qu'on souhaite trouver, et pour tout nœud x, la donnée qui s'y trouve est :

- plus grande que les données des sous-arbres gauche de x ;
- plus petite que les données des sous-arbres droit de x

Donc, on peut dire que les ABR généralisent les listes triées [1].

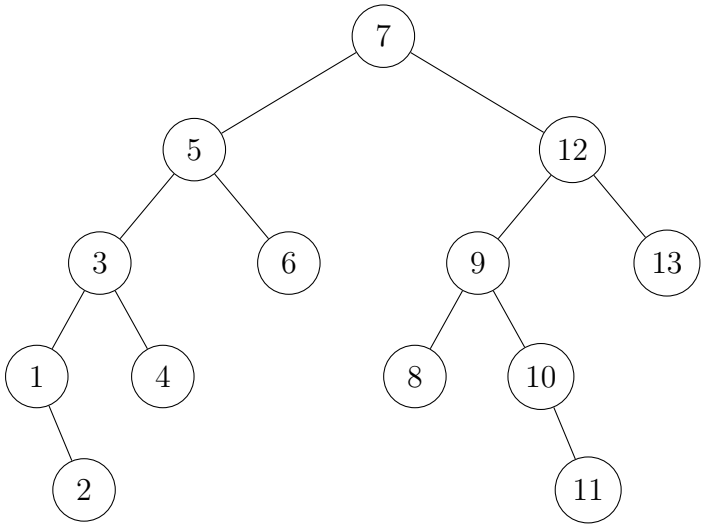

FIGURE 2 – Arbre binaire de recherche équilibré.

Source : *Code repéré en mars 2022, sur https://tex.stackexchange.com, origine de Caverac, et le code source développé figure en annexe.*

La recherche (k)

Lors de la recherche, en chaque nœud visité, on peut laisser tomber l'un des deux sous-arbres. On suit donc un chemin de la racine jusqu'à :

- Un nœud contenant k , quand k est présent dans T.
- Une référence vide, quand k est absent de T [1].

e.g. si on cherche(11), on compare 11 avec la racine 7, (11 > 7) alors on laisse tomber le sous arbre gauche et on parcourt le sous arbre droite, et on recommence le même processus récursivement jusqu'à trouver 11 (s'il existe)

Les Arbres Binaire de Recherche équilibrés de la Figure 2, garantissent un temps logarithmique dans le pire des cas.

Résumé sur les ABR

1. ABR déséquilibrés de taille n, où chaque nœud interne a un seul fils (on appelle aussi un ABR dégénéré en une liste triée), Figure 3 :
   - La recherche est en $\mathcal{O}$(h), h est la hauteur.
   - L'insertion et la suppression s'apparentent à une recherche suivie d'un travail supplémentaire.
   - Au pire cas la recherche, l'insertion et la suppression d'une donnée sont en $\mathcal{O}$(h) = $\mathcal{O}$(n).
2. ABR équilibrés de taille n :
   - Il existe plusieurs structures de données comme les AVL, les TAS, etc.
   - L'insertion et la suppression s'apparentent à une recherche suivie d'un travail.
   - Les opérations de recherche, d'insertion et de suppression, au pire cas sont en $\mathcal{O}(\log(n))$.
   - Pour un ABR aléatoire exemple Figure 2, la hauteur vaut au pire cas $\mathcal{O}(\log(n))$, car on visite de l'ordre de h = $\mathcal{O}(\log(n))$, d'où l'intérêt de travailler avec des ABR équilibrés. [1]

- La solution qui consisterait à reconstruire complètement l'arbre de façon équilibrée après chaque insertion n'est pas satisfaisante, car elle serait en $\mathcal{O}$(n), ce qui fait perdre l'avantage du $\mathcal{O}(\log(n))$. Par contre, si on commence par insérer toutes les valeurs, puis qu'on se contente de consulter très fréquemment la structure, il peut être intéressant de réaliser toutes les insertions de façon non-équilibrée dans un premier temps, puis de reconstruire l'arbre de façon équilibrée (une seule fois : le coût est alors négligeable). [3]

Voici un ABR déséquilibré :

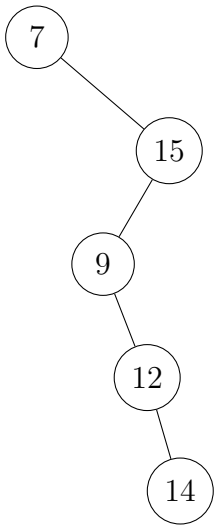

FIGURE 3 – Arbre Binaire de Recherche dégénéré en une liste triée.

Source : *Code repéré en mars 2022 sur https://tex.stackexchange.com, origine de Caverac, et le code source développé figure en annexe.*

Avantages d'ABR

- Facile à implémenter ;
- Recherche, insertion, et suppression rapides si l'ABR est équilibré.

Inconvénient d'ABR

- Rééquilibrage, *i.e.* après une suppression ou insertion on se retrouve avec un déséquilibre qui peut provoquer parfois la dégénérescence en une liste.

Complexité en temps

Pour un arbre binaire de recherche, les opérations d'insertion, de recherche et de suppression sont en $\mathcal{O}$(hauteur de l'arbre). par le théorème 7 du cours algorithmique du deuxième quadrimestre du bloc complémentaire [3]. Mais parfois dans le pire des cas, la hauteur est en $\mathcal{O}$(nombre de noeuds qu'il contient), donc en $\mathcal{O}$(n), *e.g.*, Figure 3, où on a un ABR déséquilibrés de taille n, où chaque nœud interne a un seul fils. [3]

La hauteur d'un arbre binaire équilibré est en $\mathcal{O}(\log(n))$, où n est le nombre de nœuds dans l'arbre. Et donc dans le meilleur cas pour un arbre binaire de recherche équilibré contenant n nœuds, les opérations d'insertion, de recherche et de suppression sont en $\mathcal{O}(\log(n))$. par le théorème 9 du cours algorithmique du deuxième quadrimestre du bloc complémentaire [3]

2.3 Les tables de hachage

La Figure 4 montre une table de hachage, les tables de hachage, sont des structures de données qui généralisent la notion de tableau, elles sont utilisées pour stocker un nombre de données n fini petit (k-clés à stocker) par rapport au nombre total de données potentielles (univers de données) qu'est infini, *e.g.* l'ensemble $\mathbb{R}$.

Définition

Une table de hachage est formée de d'un tableau T de taille m, et pour chacune des positions du tableau T, y a une référence qui peut être vide ou vers une liste chaînée, exemple Figure 4 on a 4 références vers des listes et 6 références vides, et les données se trouvent sur ces listes, les éléments sont stockés grâce à une fonction de hachage h ; et les éléments de la liste référencée par le tableau T sont les données qui ont la même image i par h.

Il s'agit d'un tableau ne comportant pas d'ordre (contrairement à un tableau ordinaire qui est indexé par des entiers). On accède à chaque valeur du tableau par sa clé, qu'est transformée par une fonction de hachage en une valeur de hachage (un nombre) indexe les éléments de la table, ces derniers étant appelés alvéoles (en anglais, buckets ou slots). [4]

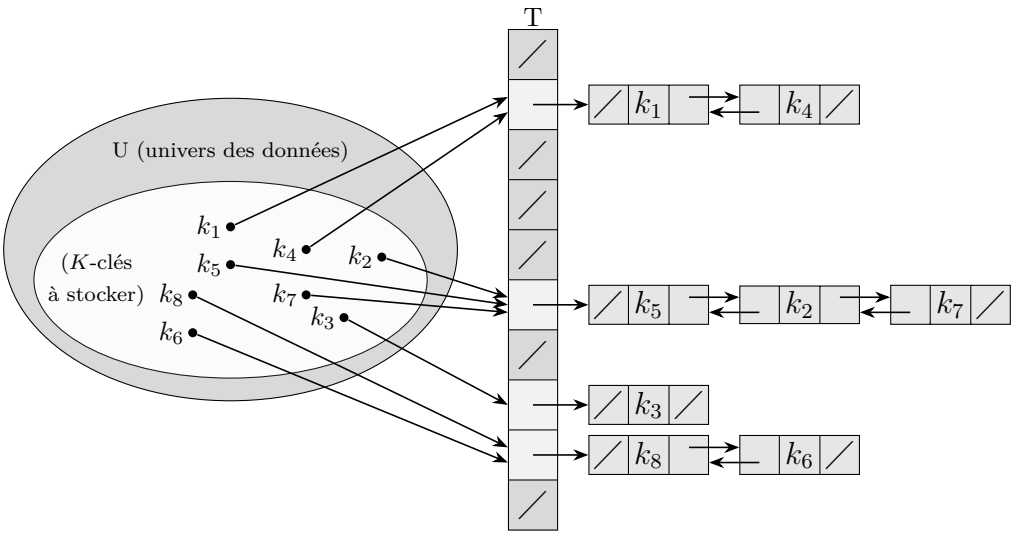

FIGURE 4 – Table de Hachage.

Source : *Repérée au cours "Rappels sur les arbres" de madame V. Bruyère, dessiné par Zarko sur lien du forum : tex.stackexchange.com, le code source développé figure en annexe.*

Avantages

1. Très facile à implémenter ;
2. Complexité excellente en moyenne en temps constant, *i.e.*, si les hypothèses suivantes se réalisent :
   - Les données sont réparties équitablement sur les listes de T (hachage uniforme) ;
   - La fonction du hachage h(k) se calcule en temps constant ;
   - le choix de m tel que $\frac{n}{m}$ est une constante. [1]

4. https://fr.wikipedia.org/wiki/Table_de_hachage

Inconvénients

- Les risques de collisions.
- Le hachage linéaire et quadratique n'utilisent que m permutations différentes sur les m ! disponible.
- Pour le hachage linéaire ou le hachage quadratique, et au fur et à mesure des insertions, on a une augmentation en nombre et en longueur de blocs de données contiguës, ce qu'on appelle le "Phénomène de blocs", *i.e.*, si on a deux données k_1 et k_2 qui ont le même point de départ, et donc on fait les mêmes sauts avant de trouver une case vide pour les insérer, car elles auront la même permutation.
- S'il y a trop de collisions, réallouer et déplacer les données.
- En pratique, c'est difficile de bien choisir h et m, si on ne connaît pas bien les données à traiter ni leur nombre n. [1]

Complexité en temps

Dans cette structure de donnée, l'accès au bon alvéole est rapide en $\mathcal{O}(1)$, grâce à la fonction de hachage qui nous renvoie l'indice adéquat dans le tableau.

Recherche

La recherche dépend de la longueur de la liste :

- Meilleur des cas est en $\mathcal{O}(1)$, quand aucune clé ayant la même adresse n'a été insérée (cas d'une recherche infructueuse), où quand la clé est en première position de la liste (cas d'une recherche fructueuse) ;
- Pire des cas est en $\mathcal{O}$(n), quand toutes les clés insérées ont la même adresse, et donc elles sont insérées dans la même liste, et la clé qu'on cherche a la même adresse ;
- En moyenne sur l'ensemble des clés recherchés, si on suppose que la fonction de hachage est uniforme, et les n clés ont été hachées équitablement dans des alvéoles, les listes auront une longueur de $\frac{n}{m}$, et le temps moyen de la recherche est en $\mathcal{O}(1+\frac{n}{m})$, (le 1 c'est pour l'accès à l'alvéole)[11].

L'insertion et la suppression se font comme dans une liste chaînée. Trouver l'alvéole est en $\mathcal{O}(1)$ et tester l'existence est en O(m/n) (en moyenne sur l'ensemble des clés recherchées)[11].

Insertion

Ajouter en tête de liste est en $\mathcal{O}(1)$[11].

Suppression

Supprimer est en $\mathcal{O}(1)$ (car la suppression d'un élément dans une liste, une fois l'élément trouvé, est en temps constant) [11].

3 Skip lists

Les Skip list sont des structures de données inventées par Pugh en 1989. Elles sont des alternatives probabilistes aux arbres équilibrés. Une Skip lists est une structures de données de recherche dynamique, de recherche parce qu'elle cherche des données (appelons clés). Si on ne trouve pas l'élément (appelons noeud), on trouve assez facilement son prédécesseur et/ou son successeur, et dynamique parce qu'elle permettent des mises à jour (insertion/suppression) des clés car ce n'est pas intéressant d'avoir une structure de données statique.

3.1 Analogie

Analogie des Skip list avec le transport ferroviaire : On a vu que la complexité des listes chaînées ordonnées est en $\mathcal{O}$(n) au pire des cas, donc une liste chaînée trié ou pas, car le tri ne nous aide pas. Mais si on essaye d'améliorer les listes chaînées, nous découvrirons les Skip list car elles étendent les listes chaînées triées. En ajoutant une liste chaînée triées (L_2) au-dessus de notre liste de départ(L_1) qui contient presque les mêmes éléments, comme dans la figure suivante :

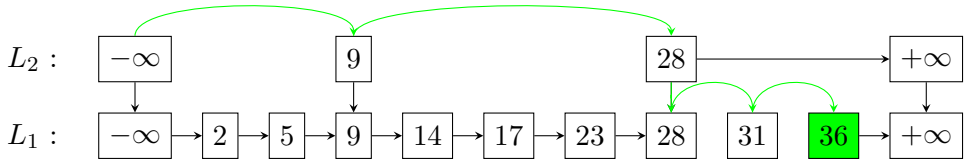

FIGURE 5 – Skip list basique avec deux listes chaînées.

Source : *Repéré en juillet 2021 sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Pour bien comprendre les Skip lists (Figure 5), prenons l'analogie des transports ferroviaires dans une grande ville comme Bruxelles, *e.g.*, on a deux lignes de transport L_1, L_2, et les arrêts sont les éléments. Les deux lignes commencent par la première station ou se trouve le nœud -∞ et le terminus est la station $+\infty$

- La première ligne, L_1 : est un métro qui dessert toutes les stations : -∞, 2, 5, 9, 14, 17, 23, 28, 31, 36, $+\infty$

- La deuxième ligne, L_2 : est un train expres qui s'arrête seulement dans les stations -∞, 9, 28 $+\infty$

Donc pour aller à la station numéro 36 on prend le train express on fait un saut vers 9, puis 28 on change et on prend le métro qui fait un arrêt 31 et puis on arrive a 36, c'est en 4 arrêts on arrive à destination, tandis que le métro s'arrête à toutes les stations avant d'atteindre notre objectif.

3.2 Définition

- h représente le nombre maximum de niveaux (qu'on va appeler la hauteur) de la Skip list S ;
- La Skip list S consiste en une série de listes $\{L_0, L_1 ... L_h\}$ telles que chaque liste L_i stocke un sous ensemble des éléments triés par ordre croissants ;
- Chaque liste a également deux nœuds (appelons sentinelles) $-\infty$ et $+\infty$, la valeur de la sentinelle $-\infty$ est plus petite que n'importe quelle clé possible de toute la liste, et celle de la sentinelle $+\infty$ est supérieure à n'importe quelle clé possible de la liste ;

- Le nœud avec la clé $-\infty$ est toujours à la position la plus à gauche et le nœud avec la clé $+\infty$ (on note parfois NULL) est toujours à la position la plus à droite ;
- Pour la visualisation, il est d'usage d'avoir la liste L_0 en bas et les listes $\{L_1, L_2 ... L_h\}$ au-dessus ;
- Chaque nœud d'une liste doit être sur un autre nœud avec la même clé en dessous. *i.e.*, si L_{i+1} a un nœud de clé k, alors L_i, L_{i-1} ,... toutes les listes en dessous contiendront la même clé k ;
- Chaque élément (qu'on appelle parfois clé) est représenté par un nœud, dont le niveau est choisi de manière aléatoire lorsque le nœud est inséré sans tenir compte du nombre d'éléments de la structure de données ;
- Un nœud de niveau i a i pointeurs vers l'avant, indexés de 1 à i ;
- Le nombre de niveaux d'une Skip list est égal au niveau maximum de la Skip list (1 si la liste est vide) ;
- Les niveaux max sont stockés dans une constante nommée MaxLevel et pas dans les nœuds.

4 Algorithmes

On va étudier les algorithmes de recherches, d'insertion et de suppression, les appliquer sur des exemples et les commenter.

4.1 Algorithme de Recherche

Prenons un exemple de recherche d'une clé dans une Skip list comme le montre la figure 6 pour la clé 24 :

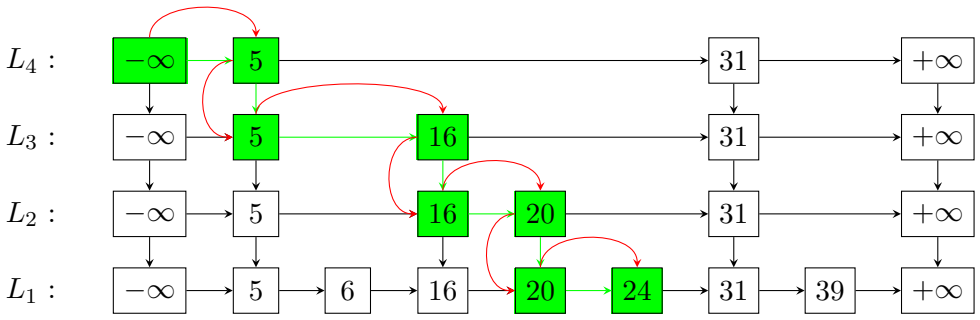

FIGURE 6 – Rechercher la clé 24.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

La recherche de 24 se fait comme suivant

- Se placer sur le niveau le plus haut L_4 ;

- l'élément suivant est 5, est-ce que $(5 \leq 24)$? Oui, se déplacer sur 5 ;

- L'élément suivant est 31, est-ce que $(31 \leq 24)$? Non, descendre d'un niveau plus bas au L_3 toujours dans noeud 5 ;

- L'élément suivant est 16, est-ce que $(16 \leq 24)$? Oui, se placer sur le nœud 16 ;

- L'élément suivant est 31, est-ce que $(31 \leq 24)$? Non, descendre d'un niveau plus bas au L_2 toujours dans noeud 16 ;

- L'élément suivant est 20, est-ce que $(20 \leq 24)$? Oui, se placer sur le nœud 20 ;

- L'élément suivant est 31, est-ce que $(31 \leq 24)$? Non, descendre d'un niveau plus bas au L_1 toujours dans noeud 20 ;

- L'élément suivant est 24, est-ce que $(24 \leq 24)$?, alors notre élément 24 a été trouvé avec succès.

Pseudo code d'algorithme de Recherche

Algorithm 1: Algorithme de Recherche dans une Skip list

```
   Entrée: Skip list liste, Entier CléRecherchée
   Sortie: Boolean Trouvé
1  Début
2      x := liste.entete
                //Parcourir les niveaux de la liste en partant du niveau haut jusqu'au niveau bas
3      Pour (i :=liste.niveau jusqu'au 1) faire
                    //Tant que la valeur du suivant du nœud courant est plus petite que la valeur
           Recherchée, on continue
4          Tant que (x.suivant[i] < CléRecherchée) faire
5              x := x.suivant[i]
6      x := x.suivant[1]
                                          //Si élément trouvé renvoyer Trouvé sinon Non Trouvé
7      Si (x = CléRecherchée) Alors
8          retourner Trouvé
9      Sinon
10         retourner Non Trouvé
```

Commentaires

La recherche se fait sur deux plans horizontal (on avance vers la droite sur le même niveau) et vertical (descendre du haut vers le bas dans le même nœud) ;

On commence toujours à parcourir les éléments à partir de l'élément gauche (la sentinelle) du niveau le plus haut de la Skip list ;

Tant que la valeur à chercher est supérieure à la valeur du nœud courant ;

Se déplacer vers la droite ;

Si la valeur est trouvée renvoyer la valeur ;

Sinon, renvoyer échec.

4.2 Algorithme d'insertion

Calculer le niveau aléatoire
Lors de l'insertion des clés, le niveau de chaque nœud est généré aléatoirement, et pour cela on a le petit algorithme suivant :

Algorithm 2: Calculer un niveau aléatoire

```
1 Sortie: Entier nouveauNiveau
2 randomLevel()
                //random() renvoie une valeur aléatoire appartenant à l'intervalle [0...1)
3 nouveauNiveau := 1
4 Tant que random() < p faire
5 |  nouveauNiveau := nouveauNiveau + 1
6 retourner min(nouveauNiveau,MaxLevel)
```

Commentaires

- MaxLevel est le niveau maximum de la Skip list.
- Cet algorithme nous garantit un niveau aléatoire jamais plus grand que MaxLevel.

Insertion(18)

Pour insérer une clé nous cherchons tout simplement l'endroit approprié, nous l'insérons et mettons à jour les pointeurs, comme le montre les figures 7 et 8, e.g : Prenons la Skip list précédente, et essayons d'inserer la clé 18, le tableau update est maintenu de sorte que lorsque la recherche est terminée, update contient un pointeur de l'élément droit du niveau i ou supérieur qui se trouve à gauche de l'emplacement [6].

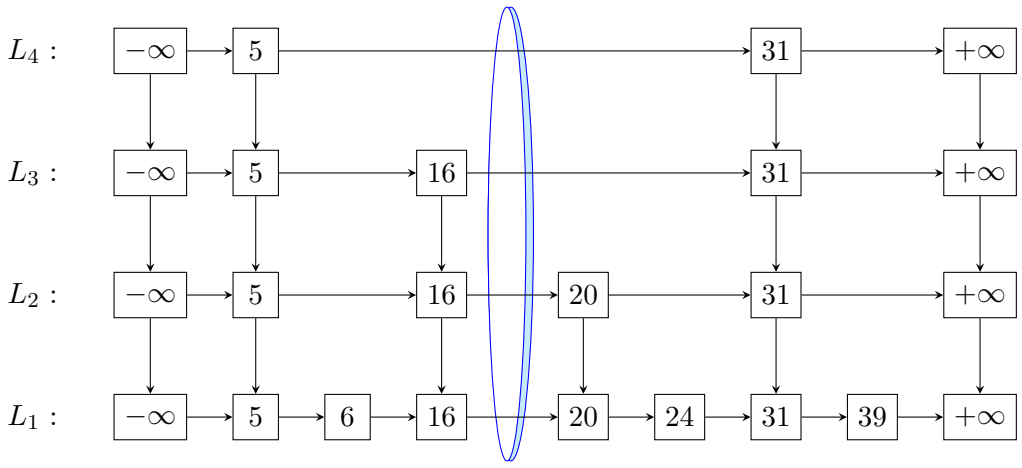

FIGURE 7 – Positionnement où la clé 18 va être insérée.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Pour chaque nouvelle insertion, on lance une pièce d'argent pour décider aléatoirement le niveau de l'élément. Si on a, pile, on monte d'un niveau, sinon on arrête. Et on continue à lancer une pièce jusqu'à ce qu'on obtienne face et on s'arrête. Si on génère un nœud avec un niveau supérieur au niveau maximum précédent de la Skip list, nous mettons à jour le niveau maximum (MaxLevel), et initialisons les parties appropriées du tableau update des pointeurs [6].

Résultat après insertion de la clé 18

Dans notre exemple figure 8, on suppose que nous avons lancé une pièce d'argent équilibrée 5 fois de suite, et on a eu 4 fois face, et 1 fois pile.

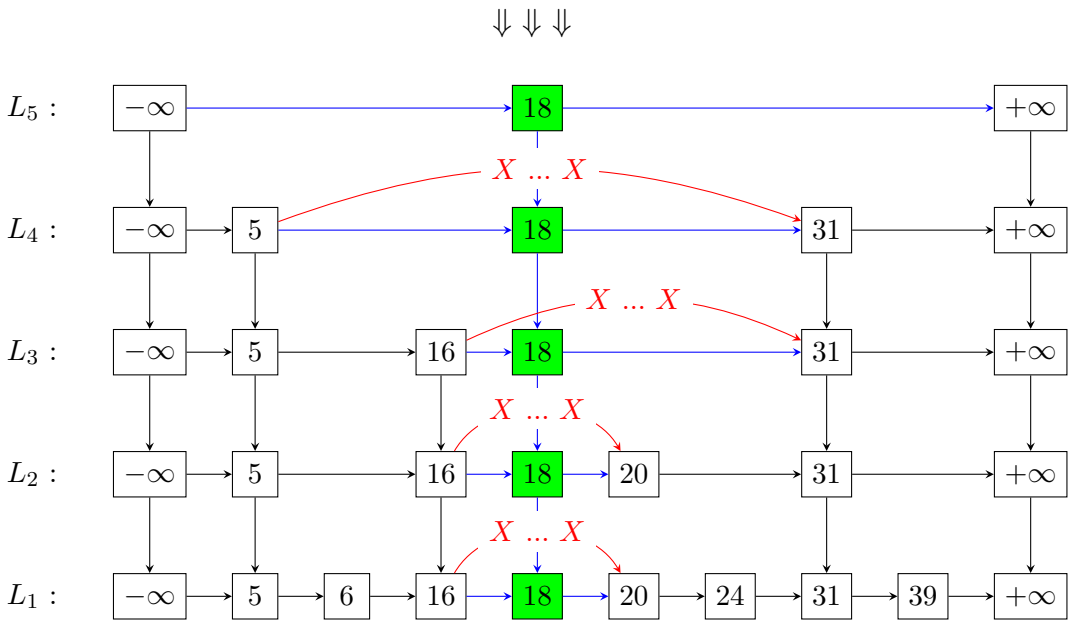

FIGURE 8 – Insertion de la clé 18, et mise à jours des pointeurs.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Commentaires

1. On commence toujours à parcourir les éléments à partir de l'entete de la sentinelle gauche du niveau le plus haut de la Skip list jusqu'au niveau le plus bas ;
2. Tant que la clé du nœud suivant est inférieure à la clé qu'on veut insérer, ou nous sommes pas arrivés à la fin, on continue à avancer sur le même niveau ;
3. Sinon on stock le pointeur dans le tableau update, et on descend d'un niveau s'il existe ;
4. On répète dans une boucle le même processus vu avant lors de la recherche de la clé 24. Aller vers la droite puis vers le bas jusqu'au niveau le plus bas où on trouve l'endroit souhaité pour insérer la clé ;
5. Si la valeur n'existe pas on procède à l'insertion comme suit :
   - On génère une hauteur aléatoire avec l'algorithme 2 : "Calculer un niveau aléatoire" ;
   - Si la hauteur aléatoire est plus grande que l'ancienne hauteur ;
     i) Dans une boucle on parcourt tous les niveaux allant de (l'ancien + 1) jusqu'au la nouvelle hauteur ;
     ii) mettre à jours le tableau update des pointeurs ;
   - On change la hauteur avec la nouvelle hauteur générée ;
   - On crée un nouveau nœud ;
   - Dans une boucle on parcourt une dernière fois les niveaux allant de 1 jusqu'à la nouvelle hauteur aléatoire ;
     i) on insère la nouvelle clé et on met à jour le tableau des pointeurs.

Voici un exemple de tableau update après insertion de la valeur 18 :

Niveau	1	2	3	4	5
Pointeur	16	16	16	5	-∞

TABLE 1 – Tableau update

Pseudo code d'algorithme d'insertion

Algorithm 3: Algorithme d'insertion

Entrée: (Skip list Liste, Entier nouvelleValeur)
Sortie: Skip list Liste

```
1  Début
2      Création du tableau update [1 ... Maxlevel]
3      x := liste.entete
                //Parcourir les niveaux de la liste en partant du niveau haut jusqu'à niveau bas
4      Pour (i :=liste.niveau jusqu'à 1) faire
                  //Tant que la valeur du suivant du nœud courant est plus petite que la valeur
           Recherchée, on continue
5          Tant que (x.suivant[i] < nouvelleValeur) faire
6              x := x.suivant[i]
7          x :=x.suivant[1]
                                  //Mise à jours du pointeur courant x dans le tableau update
8          update[i] := x
                                                  //Si le nœud exite déjà dans la liste
9      Si (x = nouvelleValeur) Alors
10         x.value = nouvelleValeur
                                                                          //L'insertion
11     Sinon
                                                        //Créer une hauteur aléatoirement
12         niveauAléatoire := randomLevel()
13         Si (niveauAléatoire > liste.niveau) Alors
14             Pour (i := liste.niveau + 1 jusqu'à niveauAléatoire ) faire
15                 update[i] = liste.entete
                                                    //Mettre à jour la hauteur de la liste
16             liste.niveau = niveauAléatoire
                                                                  //Créer un nouvau nœud
17         x := Créer nœud (niveauAléatoire, nouvelleValeur)
                                  //Insertion du nouveau nœud et réarrangement des pointeurs.
18         Pour (i :=1 jusqu'à niveauAléatoire) faire
19             x.suivant[i] = update[i].suivant[i]
20             update[i].suivant[i] := x
```

4.3 Algorithme de suppression

Commentaires

1. Pour supprimer une Clé il faut d'abord la chercher. On commence à parcourir les éléments à partir de l'entête de la sentinelle gauche du niveau le plus haut de la Skip list jusqu'au niveau le plus bas ;
2. Tant que la clé du nœud suivant est inférieure à la clé qu'on veut supprimer, ou nous sommes pas arrivés à la fin, on continue à avancer sur le même niveau ;
3. Si la valeur de l'élément suivant est plus petite que la clé à supprimer, alors se placer sur cet élément suivant ;
4. Sinon on stock le pointeur dans le tableau update, et on descend d'un niveau s'il existe ;
5. On répète le même processus vu au point 2. Aller vers la droite puis vers le bas jusqu'à niveau le plus bas où on trouve la clé à supprimer ;
6. Si la clé à supprimer n'existe pas, arrêter et sortir du programme ;
7. Sinon on procède à la suppression comme suit :
   - On parcourt dans une boucle les niveaux allant du niveau le plus bas vers le plus haut, et on réajuste les pointeurs ;
   - Supprimer le nœud souhaité ;
   - Retirer les niveaux qui ne contiennent pas d'éléments comme suit : i) Tant que le niveau de la liste est strictement plus grand que 1, et le suivant de l'entête de list pointe vers NULL, on décrémente de 1 le niveau de la Skip list.

Exemple de suppression du nœud (16) qui a la plus grande hauteur, avant :

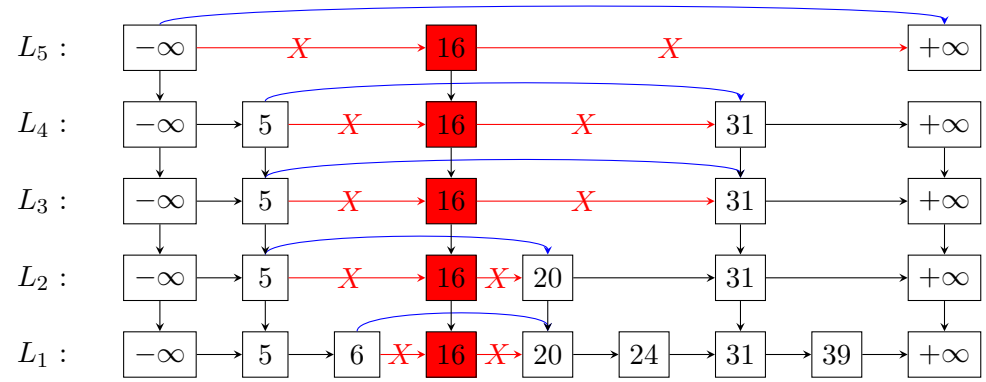

FIGURE 9 – Localisation de la clé 16 avant la suppression.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Après la suppression du nœud (16) :

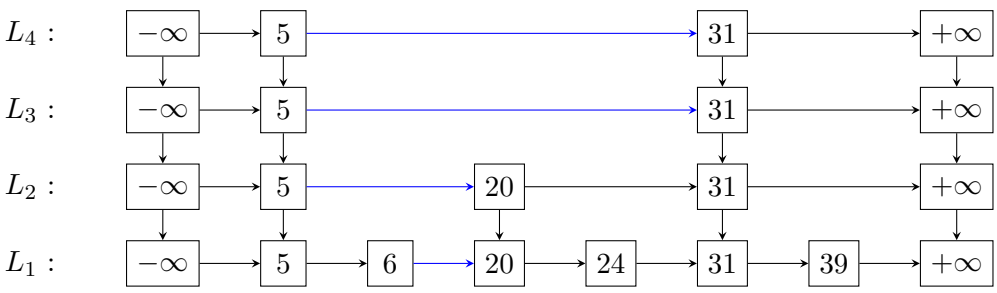

FIGURE 10 – Suppression de la clé 16, et mise à jours des pointeurs.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Pseudo code d'algorithme de suppression

Algorithm 4: Algorithme de Suppression

Entrée: (Skip list liste, Entier CléÀSupprimer)
Sortie: Skip list Liste

```
1  Début
2      Création du tableau update [1 ... Maxlevel]
3      x := liste.entete
           //Parcourir les niveaux de la liste en partant du niveau haut jusqu'au niveau bas
4      Pour (i :=liste.niveau jusqu'à 1) faire
           //Tant que la clé du suivant du nœud courant est plus petite que la clé Recherchée,
           on continue
5          Tant que (x.suivant[i] < nouvelleValeur) faire
6              x := x.suivant[i]
                                  //mise à jours du pointeur courant x dans le tableau update
7          update[i] := x
8      x :=x.suivant[1]                          //La suppression de la clé clé à supprimer
9      Si (x.clé = CléÀSupprimer) Alors
           //commencer au niveau le plus bas et réajuster les pointeurs pour supprimer le nœud
           souhaité
10         Pour (i :=1 jusqu'à liste.niveau) faire
                                  //Si l'élément suivant n'est pas le bon, quitter la boucle
11             Si (update[i].suivant[i] != x) Alors
12                 break
                                                //Supprimer x du tableau des niveaux
13             update[i].suivant[i] := x.suivant[i]
                                                                    //Suppression
14             free(x)
                                      //Retirer les niveaux qui ne contiennent pas d'éléments
15             Tant que (liste.level > 1) and (list.entete.suivant[liste.niveau] = NULL) faire
                                                    //decrementer la liste des hauteurs
16                 liste.niveau := liste.niveau – 1
```

5 Analyses

5.1 Skip lists uniformes

Coût de la recherche

Prenons l'exemple suivant qui représente une Skip List formée de deux listes chaînées L_1 et L_2, tous les éléments de L_2 sont répartis d'une manière **uniforme** comme le montre la figure 10. c'est ce qu'on appelle **Une Skip list uniforme**. Pour chercher un élément donné on parcoure la liste du haut L_2 et on parcours seulement une partie de la liste du bas L_1, on peut avoir un coût de recherche égal approximativement à $|L_1|$ dans le pire des cas.
Et donc le coût de recherche est [9] :

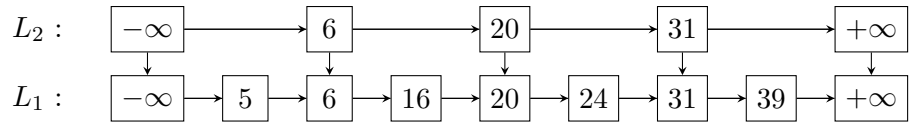

FIGURE 11 – Skip list uniforme avec deux listes chaînées.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Démonstration.

$$\text{Coût de la recherche au pire cas} = |L_2| + \frac{|L_1|}{|L_2|}$$

Et notre objectif de minimisation, est atteint quand :

$$\begin{aligned} |L_2| &= \frac{|L_1|}{|L_2|} \\ |L_2|^2 &= |L_1| = n \\ |L_2| &= \sqrt{n} \\ \text{Donc, le coût de la recherche} &= 2.\sqrt{n} \end{aligned}$$

□

Généralisation

Toujours dans le cadre de l'amélioration du coût de la recherche, maintenant on ajoute d'autres niveaux, *i.e.*, d'autres listes en haut de nos deux listes chaînées de départ, figure 10.

Hypothèse

Supposons que tous les éléments sont parfaitement répartie sur les listes du haut comme le montre la figure 10. Cette Hypothèse disparaîtra après dans le cas des analyses des Skip lists aléatoires, car il va falloir lancer une pièce de monnaie pour déterminer la hauteur.

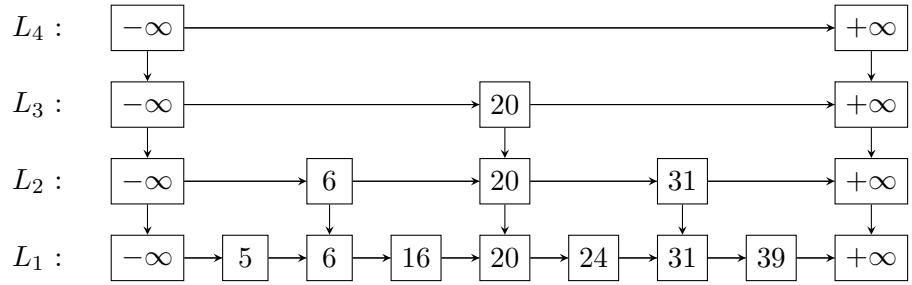

FIGURE 12 – Skip lists uniforme cas général.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Rappel 1: Identités logarithmiques

- $\forall r \in \mathbb{R}, \log_a(a^r) = r$

Pour (a, b) strictement positifs, on a :

- $\log_a b = \dfrac{1}{\log_b a}$
- $a^{\log_a b} = b$

On a vu précédemment le coût de la recherche :
Pour 2 niveaux de la Skip lists on a $\Longrightarrow |L_2| = 2.\sqrt[2]{n}$, et donc,
Pour 3 niveaux de la Skip lists on a $\Longrightarrow |L_3| = 3.\sqrt[3]{n}$
... et ainsi de suite ...
Pour k-éme niveaux de la Skip lists on a $\Longrightarrow |L_k| = k.\sqrt[k]{n}$
Pour $\log_2 n$ [1] niveaux de la Skip lists on a
$\Longrightarrow |L_{\log_2 n}| = \log_2 n.\ \sqrt[\log_2 n]{n} = \log_2 n.n^{\dfrac{1}{\log_2 n}}$

Démonstration.

$$\begin{aligned} |L_{\log_2 n}| &= \log_2 n * n^{\frac{1}{\log_2 n}} \\ &= \log_2 n * n^{\log_n 2} \\ &= 2 * \log_2 n \\ &\approx \mathcal{O}(\log n) \end{aligned}$$

□

5.2 Skip lists aléatoires

Une Skip list aléatoire comme la montre la figure 13, a des éléments répartis au hasard sur les niveaux supérieurs, c'est parce que la fonction d'insertion des éléments contient une boucle où on lance une pièce d'argent, si on a une face alors promouvoir x sinon on s'arrête, et on ajoute cet élément.

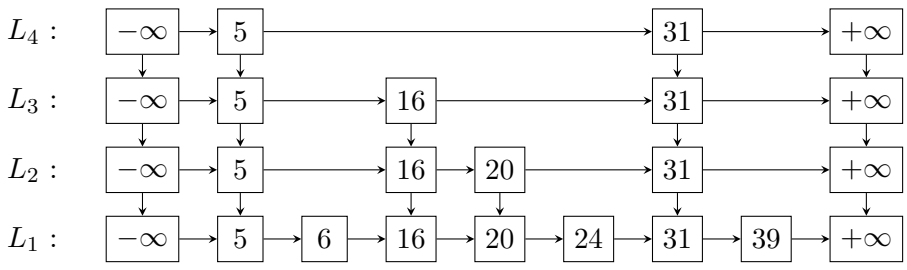

FIGURE 13 – Une Skip list aléatoire.

1. $\log_2 n$ comme hauteur a pour but de comparer les Skip lists avec les ABR équilibrés

Source : *Code repéré sur github.com via ce lien, origine de Ming-Ho Yee, le code source développé figure en annexe.*

5.3 La hauteur

Intéressons-nous maintenant à la hauteur moyenne h de la Skip list.
D'après l'algorithme d'insertion, chaque fois qu'on veut insérer un nœud dans la Skip list, on appelle la fonction **random()** qui renvoie une valeur aléatoire appartenant à l'intervalle [0,1], si on a :
- pile, c'est la probabilité d'une réussite qui vaut (1-p) ;
- face, c'est la probabilité d'un échec qui vaut (p).

Question : En combien de lancers de pièces équilibrées, allons-nous promouvoir un élément ?

Rappel 2: Rappel sur les Séries géométriques

On sait pour une série géométrique convergente réelle, avec une raison $p \in \mathbb{R}$, $|p| < 1$, sa somme vaut [a] *:*

$$\sum_{i=0}^{+\infty} p^i = \frac{1}{1-p}$$

Donc, on peut en déduire la somme suivante :
$(1p^0 + 2p^1 + 3p^2 + 4p^3 + \ldots) =$
$[(1p^0 + p^1 \ + p^2 \ + p^3 + ...)$
$\quad (+p^1 \ + p^2 \ + p^3 + ...)$
$\quad\quad (+p^2 \ + p^3 + ...)$
$\quad\quad\quad (+p^3 + ...)$
$\quad\quad\quad\quad .....]$
$(1p^0 + 2p^1 + 3p^2 + 4p^3 + \ldots) =$
$p^0(p^0 + p^1 + p^2 + p^3 + ...) + p^1(p^0 + p^1 + p^2 + ...) + p^2(p^0 + p^1 + ...) + ...$
Donc,
$(1p^0 + 2p^1 + 3p^2 + 4p^3 + \ldots) = p^0(\frac{1}{1-p}) + p^1(\frac{1}{1-p}) + p^2(\frac{1}{1-p}) + ...$
$= (\frac{1}{1-p})(1p^0 + 2p^1 + 3p^2 + 4p^3 + \ldots)$
$= (\frac{1}{1-p})^2$

a. Source : fr.wikipedia.org/wiki/Série_géométrique

Réponse :

Démonstration.

En moyenne le nombre attendu de lancers d'une pièce équilibrée, jusqu'à ce qu'on retrouve face va être la sommation suivante :

Niveau 1 : lancer la pièce 1 fois => 0 échec p et 1 réussite (1-p)
Niveau 2 : lancer la pièce 2 fois => 1 échecs p et 1 réussites (1-p)
Niveau 3 : lancer la pièce 3 fois => 2 échecs p et 1 réussites (1-p)
...
Niveau h : lancer la pièces h fois => (h-1) échecs p et 1 réussites (1-p)

Donc en moyenne le nombre de lancers de pièce est l'espérance $E(x_j)$, x_j est le nombre de lancers de pièce attendus jusqu'à ce qu'on trouve pile, c'est la sommation de :
(1-p) si on a pile dès le premier coup, donc ça sera une fois,
et puis deux fois (2) : une fois face (p), et une fois pile (1-p),
et puis trois fois (3) : deux fois face (p^2), et une fois pile (1-p), *etc.*
Et donc l'espérance $E(x_j)$ va être la sommation suivante : nombre de lancers de pièce attendus jusqu'à ce qu'on trouve pile :

$$\begin{aligned} E(x_j) &= [(1-p) + 2p^1(1-p) + 3p^2(1-p) + \ldots] \\ &= [(1-p) + 2p^1(1-p) + 3p^2(1-p) + \ldots] \\ &= (1-p) * (1p^0 + 2p^1 + 3p^2 + 4p^3 + \ldots) \\ &= (1-p) * (\frac{1}{1-p})^2 \\ &= \frac{1}{1-p} \end{aligned}$$

□

Donc en moyenne la hauteur de chaque nœud est $\frac{1}{1-p}$

La probabilité pour dépasser la hauteur h

La probabilité qu'une Skip list avec n nœuds ait une hauteur au moins h est :

$$\begin{aligned} p^h.[(1-p) + 2p^1(1-p) + 3p^2(1-p) + \ldots] &= p^h.(1-p).\frac{1}{1-p} \\ &= p^h \end{aligned}$$

Donc la probabilité pour qu'un nœud de la Skip list ait la hauteur au moins h c'est p^h.

La probabilité pour une Skip list de n nœuds ait une hauteur d'au moins h

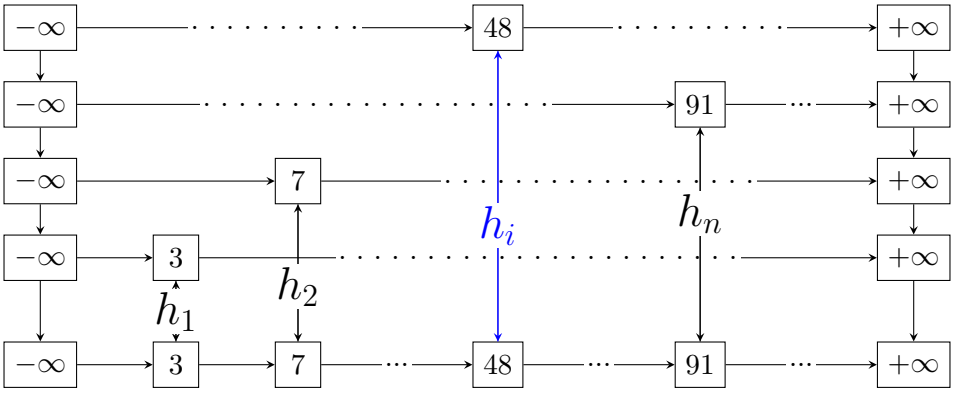

FIGURE 14 – Hauteur d'une Skip list.

Source : *Repéré sur www.github.com, origine de Ming-Ho Yee, le code source développé figure en annexe.*

Supposons qu'on a une Skip list, figure 14, avec n nœuds ce qui voudrait dire qu'on a déjà fait n insertions auparavant, et chaque nœud a atteint une certaine hauteur. Appelons ces hauteurs $(h_1, h_2, ..., h_i, ..., h_n)$, et donc il y a un nœud *e.g.*, (48) avec la plus grande hauteur (h_i).

5.4 Avec une forte probabilité

Dans les algorithmes randomisés la notion "avec une forte probabilité" ; est une notion technique très puissante qui peut être utilisée, pour notre algorithme [2].

Theorem 1

Avec une forte probabilité, le coût de la recherche dans chaque Skip lists de n nœuds est en $\mathcal{O}(\log_2 n)$ [2].

Définition informelle : Un événement E se produit avec une forte probabilité si, $\forall \alpha \geq 1$, il existe un choix approprié de constantes pour lesquelles E se produit avec une probabilité au moins $(1 - \mathcal{O}(\frac{1}{n^\alpha}))$ [2].

Définition précise : un événement (paramétré) E_α se produit avec une forte probabilité si, $\alpha \geq 1$, E_α se produit avec une probabilité au moins $(1 - \frac{c_\alpha}{n^\alpha})$, où c_α est une constante en fonction de α [2].

Le terme $\frac{1}{n^\alpha}$ est la probabilité d'erreur.

Finalement l'idée derrière tout ça, est que la probabilité d'erreur peut être très très faible en définissant α en quelque chose de grand, par ex. 100 [2].

Question : Quelle est la probabilité qu'au moins un des nœuds n de la Skip lists (figure 13) ait dépassé la hauteur h ?

Réponse : Pour répondre à ça, on va voir l'inégalité de Boole :

Rappel 3: l'inégalité de Boole

Il est bien connu que, pour deux événements E_1 et E_2, la probabilité de leur union est plus petite ou égale à la somme des deux probabilités.

$$P(E_1 \cap E_2) \leq P(E_1) + P(E_2)$$

En d'autres mots, la mesure de probabilité est sous-additive :

$$P(\cup_{i=1}^{n} E_i) \leq \sum_{1}^{n} P(E_i) \text{ [7]}$$

La hauteur de la Skip list avec une forte probabilité a c.log(n) niveaux,
$\forall n$ éléments, avec $c \geq 1$ et p=$\frac{1}{2}$

Démonstration.

La probabilité pour qu'un élément particulier fasse partie de plus de $c.\log_2(n)$ niveaux est :

$$P(\text{Un élément dans plus de } c.\log_2(n) \text{ niveaux}) = \frac{1}{2^{c.\log_2(n)}}$$

Et d'après le Rappel 1 des identités logarithmiques : $a^{\log_a(b)} = b$

$$P(\text{Un élément dans plus de } c.\log_2(n) \text{ niveaux}) = \frac{1}{n^c}$$

Et appliquons l'inégalité de Boole, pour calculer la probabilité pour n'importe quel élément fait partie de plus de $c.\log_2(n)$ niveaux.

$$\begin{aligned} &P(\text{Tout élément dans plus de } c.\log_2(n) \text{ niveaux}) \leq \\ &n * P(\text{Un élément dans plus de } c.\log_2(n) \text{ niveaux}) \\ &\leq n * \frac{1}{n^c} \\ &\leq \frac{1}{n^{c-1}} \end{aligned}$$

□

Ainsi, la probabilité d'erreur est polynomialement petite, et l'exposant ($\alpha = c - 1$) peut être rendu arbitrairement grand par un choix approprié de la constante dans la limite de niveau de $\mathcal{O}(\log n)$. [2]
Avec cette forte probabilité, si n grandit cette probabilité diminue proportionnellement.

Exemple : Pour c = 3, n=1.000,
La probabilité d'erreur, $P_{erreur} = \frac{1}{1.000^2} = \frac{1}{1.000.000}$
Dans cet exemple, c'est environ une chance sur un million qu'on dépasse la hauteur.

C'est pourquoi nous n'avons généralement pas à définir de limite explicite sur la hauteur de la Skip list, par ce que la chance d'avoir une grande hauteur en lançant simplement une pièce à plusieurs reprises est vraiment très très faible.

Avec une forte probabilité la hauteur de la Skip list est logarithmique, c'est donc assez comparable aux arbres binaires de recherche équilibrés.

Avantages

- beaucoup plus simple à implementer ;
- facile à gérer que la plus part des arbres binaires de recherhe ;
- rapidité d'exécution [12].

Inconvénient

- Les garanties sont un peut plus faible dans le sens ou c'est pas des garanties au pire cas càd qui sont toujours correcte.

6 Complexité en temps

6.1 Complexité en temps

Le temps nécessaire à l'exécution des opérations de recherche, de suppression et d'insertion est dominé par le temps nécessaire à la recherche d'un élément approprié. Pour les opérations insertion et suppression, il existe un coût supplémentaire proportionnel au niveau du nœud inséré ou supprimé.

Le temps nécessaire pour trouver un élément est proportionnel à la longueur du chemin de recherche, qui est déterminée par le modèle dans lequel des éléments de différents niveaux apparaissent lorsque nous parcourons la liste. [6]

Complexité	Moyenne	Pire
Recherche	$\mathcal{O}(\log(n))$	$\mathcal{O}$(n)
Insertion	$\mathcal{O}(\log(n))$	$\mathcal{O}$(n)
Suppression	$\mathcal{O}(\log(n))$	$\mathcal{O}$(n)

Table 2 – Complexité en temps des Skip list

7 Conclusion

Une Skip list est une structure de données très efficace pour faire des recherches rapides sur une liste chaînée ordonnée. Le but est d'améliorer les listes chaînées ordonnées pour avancer plus rapidement. De cette manière, en quelques sauts on accède à l'élément recherché. Pour cela, dans la Skip list on ajoute des niveaux avec certains éléments de la liste chaînée de base. C'est-à-dire que le niveau L_1, le plus bas, contient tous les éléments de la liste chaînée de base, et le niveau supérieur contient un peu moins d'éléments, *etc.* Par exemple, s'il y a n éléments à la hauteur 1, on s'attend à obtenir n*p éléments de hauteur 2 [12]. Où p est un paramètre important dans les Skip lists. Idéalement p=$\frac{1}{2}$ [6], ainsi chaque élément à une chance sur deux de se retrouver sur le niveau supérieur. En poursuivant ce processus, on aura de moins en moins d'éléments pour les niveaux supérieurs. Des pointeurs supplémentaires devront être ajoutés pour permettre de monter ou de descendre entre les niveaux.

Les Skip lists peuvent avoir une complexité en pire des cas telle que celle des listes chaînées. Lors de l'insertion des éléments, nous pouvons éventuellement observer un comportement où tous les nœuds auront la même hauteur, ce qui est très rare par la forte probabilité.

Les Skip lists peuvent être considérées comme des alternatives aux Arbres binaires de recherche équilibrée, dans le sens ou c'est un équilibre au pire des cas mais dont les garanties sont similaires, et avoir une garantie sur la hauteur en $\mathcal{O}(\log(n))$, ça implique qu'on sait faire la recherche, l'insertion et la suppression en $\mathcal{O}(\log(n))$.

Par contre, les Skip lists sont triées et efficaces sans avoir besoin d'un rééquilibrage compliqué comme c'est le cas avec les Arbres binaires de recherche. Comme pour les TAS avec les éclatements et les fusions, ou les AVL avec les rotations simples ou double gauche/droite. Cependant, cela peut se révéler pénible à mettre en œuvre.

Les Skip lists ont fait leur apparition dans plusieurs applications comme : Apache Portable Run-time, les bases de données relationnelles comme SingleStore, les serveurs Cyrus IMAP, *etc*[5].

8 Annexe

Toutes les sources et les codes source qu'ont servi à la rédaction de ce travail se trouvent sur le document latex sur ce lien : https://github.com/YounessKazzoul/Skip-lists

5. https://en.wikipedia.org/wiki/Skip_list https ://en.wikipedia.org/wiki/Skip_list

Références

[1] Véronique Bruyère. *Rappels sur les arbres*. Consulté en premier quadrimestre 2022. Notes prises dans le cours Structures de données de Master en sciences informatiques - Université UMONS.

[2] Erik Demaine and al. Introduction to algorithms, lecture notes on skip lists.pdf. Consulté en octobre 2022. MIT 6.046J/18.410J.

[3] Gilles G EERAERTS. *ALGORITHMIQUE*. 3^e ÉDITION 2010–2011, Consulté en deuxième quadrimestre du bloc complémentaire en 2021. Notes du cours S-INFO-048 - Année préparatoire au master 60 en informatique ULB–UMons.

[4] Sylvie Hamel. Dictionnaires ordonnés et skip list. Consulté en octobre 2022. IFT2015, A2009, Université de Montréal.

[5] Jacques-Olivier Lachaud. Notes de cours info626, l3 stic informatique algorithmique avancée. Consulté en mai 2022. LAMA, Université de Savoie, url : http://www.lama.univ-savoie.fr/pagesmembres/lachaud/Cours/INFO626/Cours/notes-de-cours.pdf.

[6] William Pugh. Skip lists : A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6) :668–676, Rédaction en 1989 et consultation en otobre 2021.

[7] Jean-François Renaud. *Cours Probabilités et statistique*. Université du Québec à Montréal (UQAM), Consulté en janvier 2022.

[8] Sriram Sankaranarayanan. Trees and graphs : Basics - université du colorado of boulder. https://www.coursera.org/lecture/trees-graphs-basics/skip-lists-7Uggo. Consulté en février 2022.

[9] François Schwarzentruber. Skip lists - listes à sauts. Consulté en octobre 2022. École normale supérieure de Rennes, url : http://people.irisa.fr/Francois.Schwarzentruber/algo2/08skiplists.pdf.

[10] Dan Suthers. Probabilistic analysis and randomized algorithms. Consulté en mars 2022. University of Hawaii Information and Computer Sciences.

[11] Jean-Stéphane Varré. Cours 4 - tables de hachage. Consulté en mai 2022. Faculté des Sciences et Technologies de Lille, url : https://www.fil.univ-lille1.fr/portail/archive19-20/~varre/portail/asd/doc/cours4-2.pdf.

[12] Steve Zaretti. La skip-list. Consulté en août 2021. UMONS - url : https://steve.zaretti.be/files/skiplist.pdf.