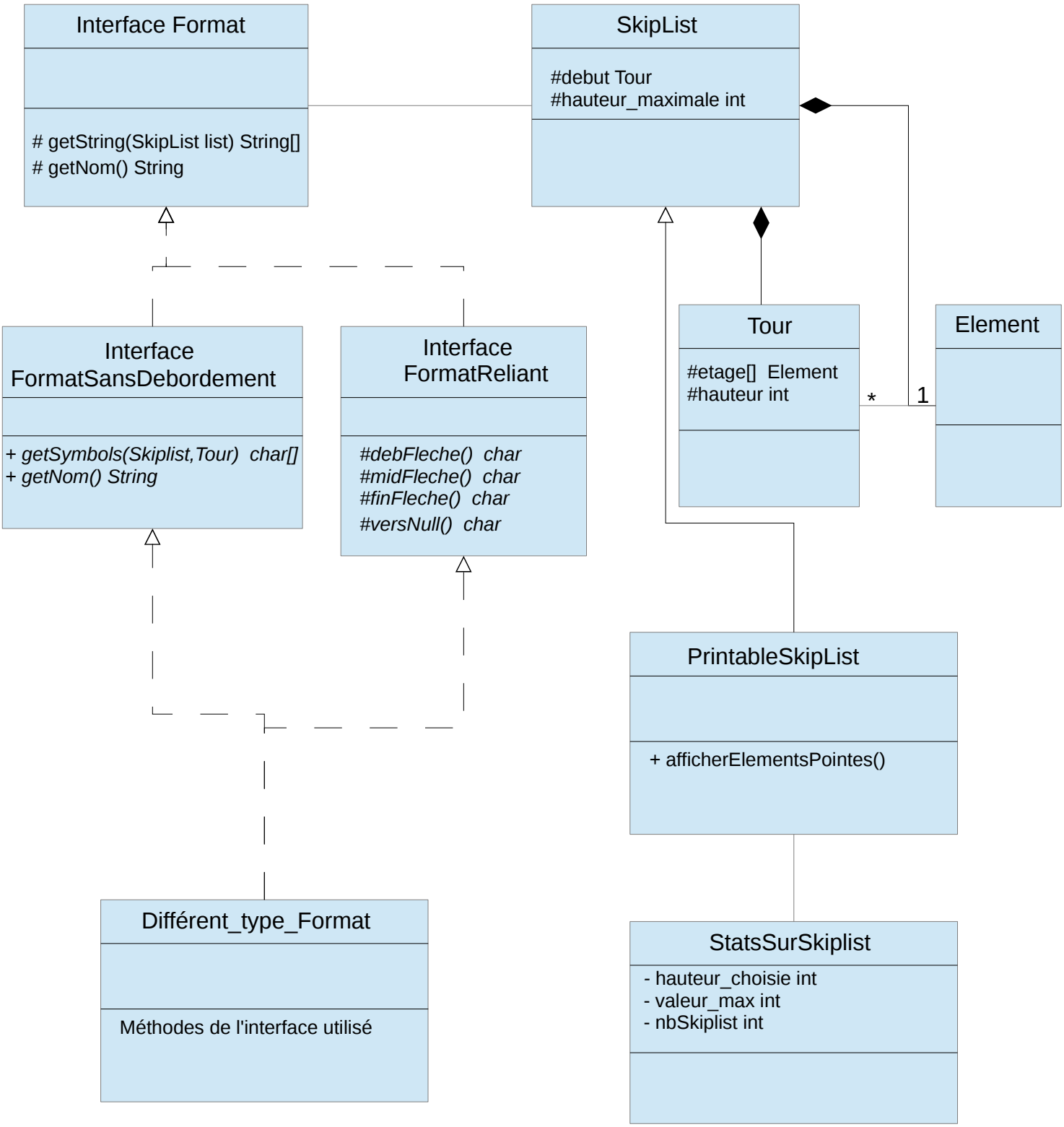
Julian BRUYAT
Julien ESTAILLARD

Devoir n°1 d'algorithmique : Skiplist

Licence 3 Informatique
Faculté des Science Aix-Montperrin
2016 - 2017

Vue d'ensemble :



Dans ce diagramme de classe nous avons retenu uniquement les points importants de chaque classes comme leurs attributs ou les méthodes les plus importantes des classes.

Les Structures :

Classe Skiplist : (fichier Skiplist.java)

Représente la liste de l'ensemble des éléments avec les tours de pointeurs permettant d'atteindre des éléments éloignés dans la liste. Il s'agit de l'implémentation basique des skiplist
Attributs :
- « tour » Tour de pointeur des premiers éléments de la liste (cette tour à la hauteur maximal : c'est la sentinelle).
- « hauteur maximal » est la hauteur max des tours de pointeurs.

Classe Element : (fichier Skiplist.java)

Représente un élément de la liste
Attributs :
- « cle » valeur de l'élément
- « tour » tour de pointeur de l'élément

Classe Tour : (fichier Skiplist.java)

Représente une tour de pointeurs vers d'autres éléments
Attributs :
- « etage » est un tableau pointant sur les éléments suivants
- « hauteur » est la hauteur de la tour

Classe PrintableSkipList : (fichier Visualisation.java)

Implémente les méthodes d'affichage et d'analyse de la skiplist sans surcharger la classe Skiplist.

Classe StatsSurSkiplist :

Réalise des calculs sur plusieurs skiplist. Permet de récupérer des données de test et d'affichage par exemple le pourcentage de niveau des tours de pointeurs dans plusieurs skiplist.

Classes Format :(fichier Visualisation.java)

Base pour implémenter les « Format » d'affichages (c'est-à-dire comment afficher la structure d'une skiplist). On a ainsi d'autress classes permettant d'afficher une skiplist de différentes manières (classes FormatSansDebordement ...).

Nous avons égalemen une classe de test permettant d'afficher l'interface permettant de tester notre implémentation des skiplist. (Fichier menu.java)

Langage de Programmation :

Nous avons choisis d'utiliser le langage Java par affinité puisque nous connaissions le mieux ce langage de programmation. Il est de plus conçu pour avoir un code assez simple à comprendre et bien structuré

Méthodes principales :

Dans la classe Skiplist :
- inserer (int cle) /inserer(Element nouvelElem)

Cette méthode permet de créer un « Element » avec une valeur « cle » et une tour de pointeur de hauteur aléatoire. Elle retourne un booléen vrai si l'élément est bien placé et faux sinon.

Cette méthode utilise une autre méthode pointerSurAuplus(Element newElement).

Dans la classe Tour :
- pointerSurAuplus (Element newElement)

Cette fonction permet de changer les pointeurs de la tour du nouvelle élément sur les bons éléments (en gros c'est ce qui permet d'insérer).

On regarde si dans la tour de l'élément en cours il y a un pointeur qui pointe sur une « cle » plus grande que la « cle » de « newElement ».

Puis on prend la hauteur minimum entre l'élément en cours et le « newElement ».

Enfin on copie les pointeurs.

Remarque : S'il n'y a pas de « cle » plus grande que la « cle » de « newElement » rien ne se passe.

Difficultés :

Une des plus grandes difficultés était de réfléchir à comment implémenter les méthodes et à leur donner des noms pour qu'elles puissent être rapidement compréhensible. La méthode « pointerSurAuPlus » a par exemple été une des plus difficiles à concevoir clairement puis à nommer précisément.

Rajouter une sentinelle (tour de pointeur) en début de liste pour pouvoir atteindre les éléments plus rapidement. Cette modification étant intervenu lorsque le programme était bien avancé nous avons dû modifié la structure de la plupart des fonction qui utilisaient le première élément comme référence. Nous avions commencé par modifier la manière de gérer le premier élément en le remplaçant par un tableau de pointeurs, puis nous nous sommes rendu compte que créer une classe tour reprenant le code de element gérant la modification des pointeurs étaient plus simple car il suffisait ensuite de remplacer le début par un objet de classe tour.

Nous avons aussi eu des difficulté a déterminer quels tests et résultats étaient pertinent. Exemple il est inutile d'afficher la répartition des tours, puisqu'une tour de hauteur h donnée pourrais ce trouvée n'importe ou dans la skiplist. Par contre comparer la skiplist avec une linkedlist est pertinent, on a comparé le temps d'exécution pour la recherche d'un élément dans les deux type de listes par exemple. Nous avons également eu l'idée de tester le temps de recherche entre une skiplist idéale et une skiplist tel qu'implémenté avec des tours de hauteur aléatoire.

Nous n'avons pas eu le temps de tester les variations de temps d'exécution en fonction de la probabilité de gagner un étage dans les tours.

Modification :

Dans la fonction de suppression d'un élément de la liste, nous testions si l'élément en question est dans la liste grâce à une fonction. Mais cela c'est révélé inutile car on demande à tout les élément de ne plus pointer sur l'élément à supprimer si il n'existe pas la fonction ne fera rien.

A la base, tout était implémenté dans la classe Skiplist. Nous avons peu à peu délégué les fonctionnaliés aux classes Elements puis Tour dans un soucis de lisibilité du code

Amélioration :

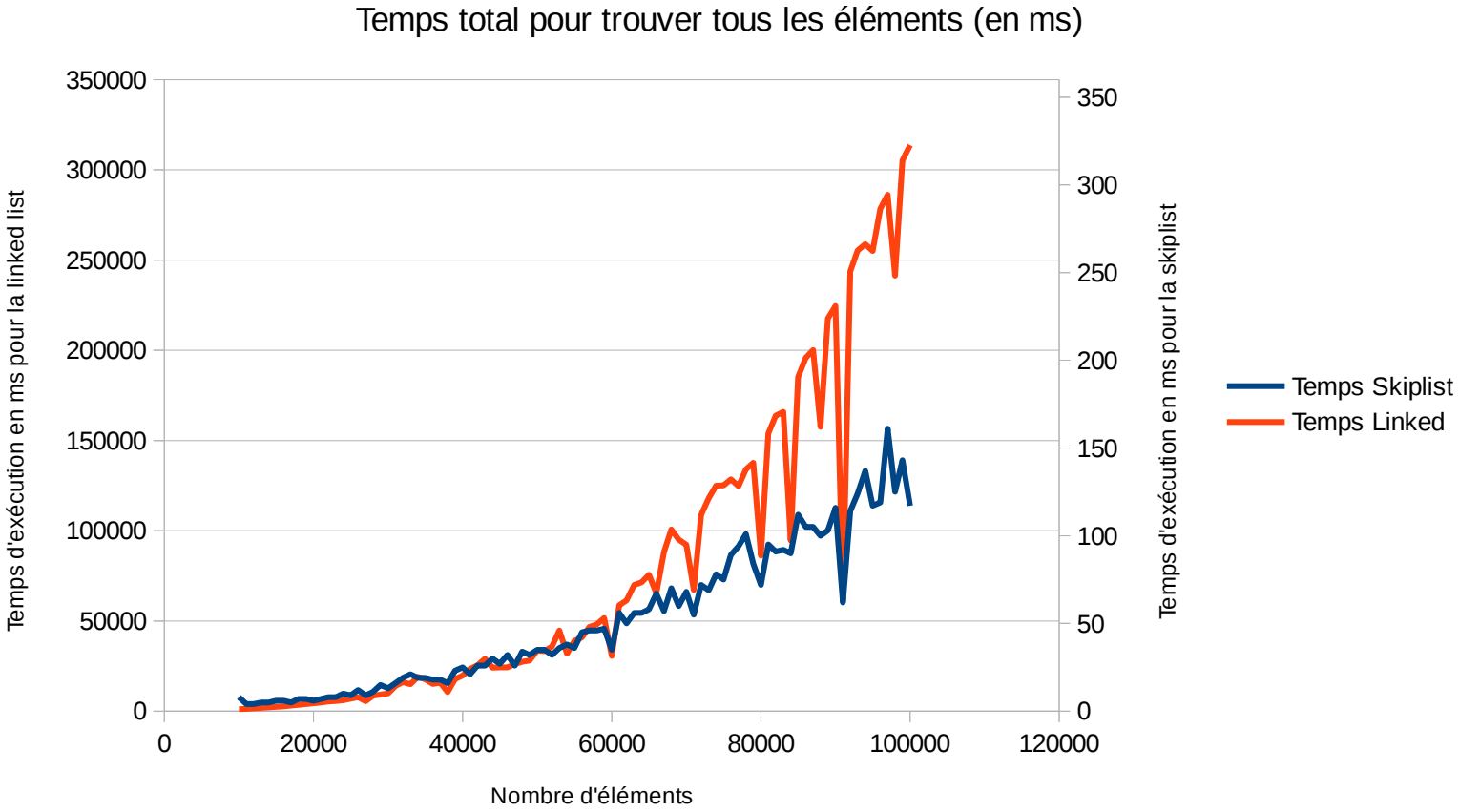
On pourrait mettre des tests sur le menu pour éviter que le programme n'échoue si l'utilisateur rentre des données farfelues.

On pourrait également penser à automatiser plus de tests afin de pouvoir générer des graphs à la manière de ce qui a été fait pour la comparaison linkedlist / skiplist, mais il faudrait pour cela trouver un moyen de contrôler le manière dont java chronomètre l'exécution d'un programme (en

bloquant l'exécution du garbage collector pendant la recherche par exemple)

Résultats :

Nous avons utilisé un fichier excel pour pouvoir interprété les données récoltées.



Sur cette exemple on va chercher tout les éléments des liste dans l'ordre d'insertion.

On recherche les éléments à la manière d'une linkedlist (en ne considérant que le premier étage de chaque tour) et à la manière d'une skiplist.

On voit que les skiplist sont beaucoup plus efficaces (rapport de 1000 avec 10000 éléments dans le temps de recherche des élémens) et que leur temps de recherche croit moins vite (on a un rapport de 2000 au bout de 10000 éléments).

Inspiré du test précédent, nous avons également comparé les temps entre un schéma de skiplist généré aléatoirement et un schéma « idéal » (tel que vu lors de la présentation des skiplist en cours).

• 11
Nombre de valeurs ? 100000
Skiplist générée aléatoirement crée
Temps d'accés avec schéma idéal : 36ms
Temps d'accés avec schéma généré aléatoirement : 51ms
• 11
Nombre de valeurs ? 1000000
Skiplist générée aléatoirement crée
• 34
Temps d'accés avec schéma idéal : 483ms
Temps d'accés avec schéma généré aléatoirement : 787ms

On peut voir que si il y a une perte de performance pour les skiplist telles qu'elles sont implémentées, celle-ci est négligeable (on trouve un coefficient multiplicateur légèrement inférieur à 2, alors que le gain par rapport à une linkedlist est d'environ 1000 pour cet ordre de grandeur d'éléments).