Devoir à la maison

Instructions générales

- Ceci est un devoir à la maison. Vous avez donc du temps pour écrire et surtout **tester** vos fonctions. Vous devez donc écrire sur un ordinateur les fonctions puis vérifier sur quelques exemples qu'elles ont bien le comportement attendu (vous pouvez en particulier créer des listes puis essayer d'insérer certaines valeurs, puis rechercher des valeurs et vérifier que celles que vous avez insérées sont trouvées par la fonction, les autres non) ;
- Il n'est pas acceptable que vous donniez des fonctions qui font des erreurs sur tous les exemples (avec des erreurs de syntaxe évidentes) ;
- La première partie est très simple. Tout le monde doit être capable de la faire. La seconde partie est nettement plus complexe, mais vous pouvez travailler en groupe pour vous aider à la faire. Les fonctions demandées sont conceptuellement difficiles mais peuvent s'écrire en peu de lignes ;
- Vous pouvez me poser des questions (dans la limite du raisonnable) par mail (`victor.poupet@lif.univ-mrs.fr`);
- Le devoir doit être rendu par mail soit sous la forme d'un unique document texte contenant les fonctions demandées ainsi que les réponses aux questions de complexité, ou bien sous la forme d'un fichier .py contenant les fonctions et d'un fichier joint contenant les réponses aux autres questions dans le format de votre choix (texte, pdf, rtf, etc.).

Dans ce devoir, nous allons étudier différentes structures de données permettant de manipuler des *ensembles d'entiers*. L'idée est de définir une structure pouvant contenir des entiers sur laquelle on puisse facilement effectuer les trois actions suivantes :
- `ajouter(x, E)` qui ajoute l'élément x dans l'ensemble E ;
- `appartient(x, E)` qui renvoie `True` si x est dans E et `False` sinon ;
- `supprimer(x, E)` qui supprime l'élément x de l'ensemble E (on suppose que x est bien dans E).

Dans un ensemble, l'ordre dans lequel on a ajouté les éléments n'est pas important, il n'est donc pas nécessaire que la structure conserve l'ordre d'insertion.

1. Listes chaînées

Une première idée très simple pour implémenter des ensembles est d'utiliser des listes chaînées, une liste représentant l'ensemble de ses éléments. Si l'on considère des listes non ordonnées, la fonction `ajouter` se contente d'ajouter le premier élément en tête de liste.

1. En utilisant les listes chaînées vues en cours, écrivez les fonctions `ajouter`, `appartient` et `supprimer`.

2. Quelle est la complexité de chacune de ces fonctions en fonction du nombre d'éléments n contenues dans la liste ?

On peut essayer d'améliorer légèrement la complexité de la fonction `appartient` dans une liste chaînée en utilisant des listes ordonnées. Dans ce cas, on suppose qu'à tout moment les éléments contenus dans la liste sont triés du plus petit (en tête de liste) au plus grand.

3. Ré-écrivez les fonction `ajouter` et `appartient` dans le cas de listes chaînées ordonnées. La fonction `ajouter` doit maintenant parcourir la liste pour trouver l'endroit où ajouter le nouvel élément afin que la liste reste ordonnée. Dans la fonction `appartient`, on peut arrêter la recherche dès qu'on rencontre un élément plus grand que x.

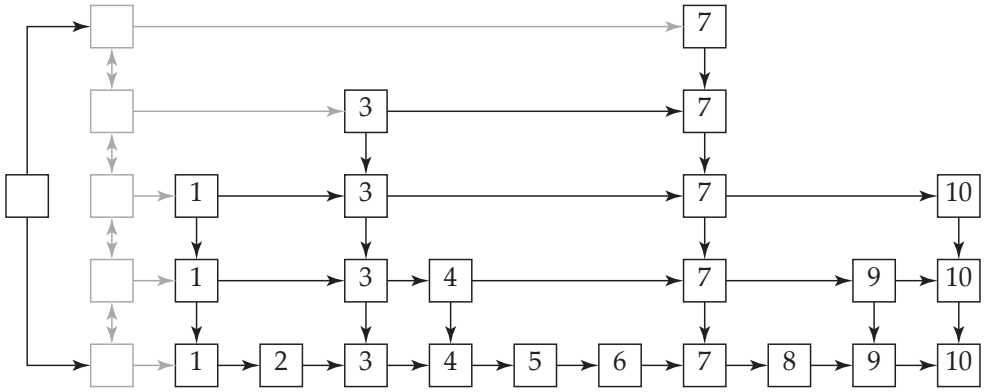

FIGURE 1 – Un exemple de *skip-list*. Au bout de chaque chaîne (verticale ou horizontale) il y a un `None` qui n'est pas représenté sur la figure.

L'utilisation de listes ordonnées ne permet pas de gagner beaucoup en complexité dans le cas de la recherche (si l'élément n'est pas dans l'ensemble, en moyenne on ne parcourt que la moitié de la liste, mais ce n'est pas significativement meilleur) et complique nettement l'insertion. peut être acceptable d'avoir une insertion en temps $O(n)$ si l'on utilise une structure de données pour laquelle on sait que l'on fera beaucoup plus d'appels à la fonction `appartient` qu'à la fonction `inserer`, mais il faudrait alors obtenir une meilleure complexité pour la fonction `appartient`.

2. *Skip-Lists*

Les *skip-lists* sont une structure de donnée introduite en 1990 pour effectuer efficacement des recherches dans des listes ordonnées. L'idée est d'enrichir les listes chaînées simples pour pouvoir avancer plus rapidement dans la liste, en sautant des éléments. Pour cela, la liste est organisée en niveaux. Le niveau 0 (le plus bas) est une liste chaînée classique, où chaque nœud pointe vers son successeur direct.
Le niveau 1 contient une certaine fraction des nœuds de la liste choisis aléatoirement avec probabilité p. Dans tout ce sujet, nous prendrons $p = \frac{1}{2}$, c'est-à-dire que chaque nœud a une chance sur deux d'appartenir au niveau 1, et donc que le niveau 1 ne contient en moyenne que la moitié des nœuds de la liste.
On répète ensuite ce processus pour chaque niveau supérieur : chaque élément du niveau 1 a une probabilité p d'être dans le niveau 2, chaque élément du niveau 2 a une probabilité p d'être dans le niveau 3, etc. Il y a donc de moins en moins d'éléments dans chaque niveau de la liste (en moyenne la moitié des éléments du niveau n sont dans le niveau $(n+1)$) et donc on finit par arriver sur un niveau ne contenant aucun nœud de la liste. On s'arrête donc à ce dernier niveau.
La figure illustre une *skip-list* contenant les entiers de 1 à 10. On voit par exemple sur cette figure que l'élément 1 est dans les trois premiers niveaux, l'élément 2 n'est que dans le premier niveau et l'élément 7 est dans les cinq niveaux de la liste.
Une *skip-list* peut donc être vue comme une liste de listes (la liste de niveau 0, la liste de niveau 1, etc.). Cependant ces listes ne sont pas indépendantes, puisque de chaque nœud de la liste de niveau $(n+1)$ on peut « descendre » directement dans la liste de niveau n (en plus des champs `valeur` et `suivant`, les nœuds ont maintenant un champ `inferieur` qui permet de descendre d'un niveau).
Pour pouvoir commencer au debut de chaque liste (de n'importe quel niveau), on ajoute des nœuds spéciaux (en gris sur la figure) ne contenant pas de valeur qui servent uniquement à indiquer le début des listes. Par la suite, on verra qu'il est parfois nécessaire de traiter les différentes listes du niveau le plus bas au niveau le plus haut, et donc ces nœuds de début de liste ont des pointeurs permettant de descendre mais aussi de monter. Chaque nœud de début a donc un champ `suivant` (qui donne le premier *vrai* nœud de la liste), un champ `superieur` et un champ `inferieur` permattant de monter

ou descendre.
Enfin, une *skip-list* est simplement donnée par les nœuds de début des listes de plus bas et plus haut niveau. Ainsi, un élément de type `SkipList` n'aura que deux champs `haut` et `bas` permettant d'accéder aux nœuds de début des deux listes extrêmes (les autres listes sont atteintes en montant ou descendant à partir des deux extrémités).
On utilisera donc les classes et constructeurs suivants (donnés dans le fichier `devoir.py`) :

```
class SkipList:
        pass
class Noeud:
        pass
class Debut:
        pass

def noeud(x):
        n = Noeud()
        n.valeur = x
        n.suivant = None   # le champ suivant permet d'avancer dans la liste
        n.inferieur = None # le champ inferieur permet de descendre d'un niveau
        return n

def debut():
        # les elements de type Debut sont des noeuds n'ayant pas de valeur places
        # en tete des listes. Il faut pouvoir monter et descendre d'un niveau a
        # partir d'un Debut.
        d = Debut()
        d.superieur = None # pour monter d'un niveau
        d.inferieur = None # pour descendre
        d.suivant = None   # pointe vers le premier vrai element de la liste
        return d

def skiplist():
        l = SkipList()
        l.haut = None   # le Debut de la liste de plus haut niveau
        l.bas = None    # le Debut de la liste de plus bas niveau (contenant tous les el
        return l
```

4. Puisque le premier niveau contient tous les éléments de la liste, et que chaque niveau contient en moyenne la moitié des éléments du niveau inférieur, combien y a-t-il en moyenne de niveaux dans une *skip-list* contenant n entiers ?

A. Insertion

L'insertion dans une *skip-list* est assez complexe. L'idée est d'insérer le nouvel élément dans les niveaux en partant du niveau le plus bas. Après chaque insertion, on décide avec probabilité p si il doit également être inséré dans le niveau supérieur. En plus d'insérer un nouveau nœud contenant l'entier à insérer dans le niveau correspondant, il faut que chacun de ces nouveaux nœuds soit relié à ceux des niveaux inférieurs (donc il faut se souvenir du nœud créé au niveau n lorsque l'on insère celui du niveau $(n+1)$).
On peut utiliser les fonctions suivantes (la fonction `filp` sert uniquement à réaliser un tirage avec probabilité $1/2$) :

```
import random
def flip():
        """Fonction permettant de tirer a pile ou face.
        Elle renvoie True ou False avec probabilite 1/2"""
        return random.choice([True, False])
```

```
def inserer(x, l):
        """Cette fonction insere un element x dans une skip-list l"""
        premier = l.bas
        inferieur = None
        test = True

        while test:
                if premier == None:
                        premier = debut()
                        premier.inferieur = l.haut
                        l.haut = premier
                        if l.bas == None:
                                l.bas = premier
                n = premier # on part du premier noeud
                while n.suivant != None and x >= n.suivant.valeur:
                        n = n.suivant
                nx = noeud(x)
                nx.suivant = n.suivant
                n.suivant = nx
                nx.inferieur = inferieur
                inferieur = nx
                premier = premier.superieur
                test = flip()
```

(la version écrite dans le fichier `devoir.py` est un peu commentée)

5. Puisqu'il faut environ p opérations pour insérer un élément dans une liste chaînée (simple) ordonnée de longueur p, et sachant qu'il y a en moyenne $n/2^k$ éléments dans le niveau k d'une *skip-list* contenant n éléments et qu'un nouvel élément doit éventuellement être inséré dans chacun des niveaux de la *skip-list*, combien faut-il d'opérations pour insérer un élément dans une *skip-list* ?
(On ne cherche pas une réponse exacte mais l'ordre de grandeur du nombre d'opérations)

B. Recherche

Pour rechercher une valeur dans une *skip-list*, on va partir de la liste de plus haut niveau (celle dont le premier vrai nœud est `l.haut.premier.suivant`) et avancer de nœud en nœud jusqu'à trouver le plus avancé dont la valeur est inférieure à x (donc on avance tant que `n.suivant.valeur` est inférieur à x ou bien jusqu'à ce que `n.suivant.valeur` soit `None`, auquel cas on est arrivé au bout de la liste).
Si la valeur du nœud atteint est x, on répond `True` car x est bien dans la liste. Sinon, on descend dans la liste inférieure (on passe donc au nœud `n.inferieur`, et on recommence à avancer dans cette nouvelle liste jusqu'à atteindre de nouveau le nœud le plus avancé dont la valeur soit inférieure à x).
On continue ainsi à descendre aux niveaux inférieurs, en avançant au maximum dans chaque niveau avant de descendre. Si on trouve x on répond `True`, si par contre on finit par descendre en dessous du dernier niveau (on atteint un nœud valant `None` en suivant un champ `inferieur`) on répond `False` car la valeur x n'est pas dans la liste.

6. Écrivez la fonction `appartient(x, l)` qui recherche la valeur x dans une *skip-list* l en parcourant les nœuds comme expliqué précédemment.

L'intérêt de la recherche en partant des niveaux élevés et en descendant progressivement (qui est tout l'intérêt des *skip-lists*) est que l'on avance beaucoup plus rapidement dans la liste en suivant les liens des niveaux supérieurs, et que l'on ne descend d'un niveau (ce qui ralentit la traversée de la liste) que lorsque l'on ne peut plus avancer dans les niveaux supérieurs (parce qu'en avançant plus vite, on dépasse la valeur x recherchée).
Pour trouver la complexité de la fonction `appartient`, il faut estimer le nombre de déplacement que l'on doit faire dans chacun des niveaux.

Le niveau le plus élevé contient toujours environ le même nombre d'éléments (si une *skip-list* a plus d'éléments, elle a plus de niveaux, mais son niveau le plus haut a toujours en moyenne 1 ou 2 éléments). Il faut donc un nombre constant de déplacements pour le traverser.
Si l'on sait que x se trouve entre deux nœuds consécutifs de la liste de niveau $(k+1)$ ayant les valeurs v_1 et v_2 (donc $v_1 < x < v_2$), puisque la liste de niveau k contient deux fois plus de nœuds, il y a en moyenne 2 nœuds entre les valeurs v_1 et v_2 dans la liste de niveau k (s'il y en avait beaucoup plus, un de ces nœuds serait aussi dans le niveau $(k+1)$). Il faut donc traverser 1 ou 2 nœuds (en moyenne) dans le niveau k avant de descendre au niveau inférieur.

7. Si l'on n'effectue en moyenne qu'un nombre constant (un ou deux) de déplacements dans chaque niveau de la *skip-list*, combien de déplacements doit-on faire au total pour trouver si x appartient ou non à la liste ? En déduire la complexité de la fonction `appartient`.

C. Suppression

8. Si vous en avez le courage, écrivez la fonction `supprimer(x, l)` qui enlève une valeur x d'une *skip-list* avec une complexité $O(\log_2(n))$.
Remarque : Cette fonction ressemble à la recherche, mais lorsque l'élément est trouvé il faut le supprimer dans chacun des niveaux en reliant directement le nœud précédant celui qui contient x à son successeur.

D. Bilan

La complexité de recherche dans une *skip-list* est nettement meilleure que dans le cas d'une liste chaînée (ordonnée ou non). C'est la même complexité que celle qu'on obtient sur les arbres binaires de recherche lorsque ceux-ci sont bien équilibrés (lorsque la hauteur de l'arbre est logarithmique en le nombre de nœuds).
Cependant, dans le cas des arbres binaires de recherche, si l'on n'a aucune hypothèse sur l'ordre dans lequel les valeurs sont ajoutées dans l'arbre, le seul moyen d'assurer que l'arbre reste équilibré est de corriger la structure à chaque insertion à l'aide de rotations (comme vu en TD avec les arbres équilibrés, ou par la technique des arbres *rouge-noir*). Ces techniques de rotation sont assez fastidieuses à coder correctement et les *skip-lists* sont censées proposer une alternative plus simple (il est possible que la *skip-list* n'ait pas une bonne répartition dans chaque niveau, mais ça ne dépend que des tirages aléatoires fait en chaque nœud et non pas de l'ordre dans lequel les éléments sont insérés, ce qui facilite les preuves probabilistes).
Pour ce qui est de l'insertion, l'insertion dans un arbre binaire de recherche est de complexité $O(\log_2(n))$ (même si l'on corrige la structure avec des rotations). L'insertion que l'on a écrite ici est linéaire, mais on peut l'améliorer en utilisant un parcours semblable à celui fait pour la recherche et la suppression qui permet de ne pas reparcourir chacun des niveaux depuis le début. On arrive alors également à une complexité $O(\log_2(n))$.