DM : Structures de données et complexité

Exercice 1 (Une liste avec un accès plus rapide)**.**
Bob est un étudiant qui suit un cours d'algorithmique, et qui aime beaucoup la structure de liste. Il trouve formidable le fait de pouvoir insérer un élément en tête de liste en temps constant, mais il est frustré par le coût $\Theta(i)$ nécessaire pour accéder au i-ème élément d'une liste. Il décide donc d'enrichir la structure de liste afin d'améliorer cette dernière complexité.

Bob a une idée pour cela. Afin d'accélérer l'accès au i-ème élément d'une liste `lst`, il propose de construire une deuxième liste plus courte (et donc plus rapide à parcourir), contenant des pointeurs vers des cases régulièrement espacées de la première liste. Il fixe un entier k (sans trop savoir pour l'instant quelle valeur choisir pour k), et crée pour chaque case `c` à distance $i \times k$ (pour tout entier $i \geq 0$) de la fin de la liste une case jumelle, de type `bobcase`, pointant vers `c`. Il renseigne la position de la première case et de la première `bobcase` dans une structure qu'il baptise `boblist`. Voici la définition de cette structure, et une illustration en est donnée dans la Figure 1

```
structure boblist :
    n : entier fixé        // taille de la boblist
    k : entier fixé        // longueur du saut des bobcases
    head : case            // pointeur vers la première case, initialisé à NULL
    bobhead : bobcase      // pointeur vers la première bobcase, initialisé à NULL

structure case :
    val : valeur           // valeur dans la case
    next : case            // pointeur vers la case suivante

structure bobcase :
    twin : case            // pointeur vers la case jumelle
    bobnext : bobcase      // pointeur vers la bobcase suivante
```

Dans la suite on pourra utiliser les constructeurs `new boblist()`, `new case()`, `new bobcase()` qui construisent respectivement une boblist, une case, et une bobcase vides.

1. Écrire un algorithme `getElement(lst,`i`)` prenant en entrée une boblist `lst` de taille n et une position $i \in \{0, \ldots, n-1\}$, et qui renvoie la valeur de la i-ème `case` de `lst` en suivant le moins de pointeurs `suiv` possible. Calculer sa complexité en fonction de i et k. — 3 pts
2. Écrire un algorithme `insert(lst,`x`)` de complexité $O(1)$ prenant en entrée une boblist `lst` et qui insère une nouvelle `case` de valeur `x` au début de `lst`. Attention à bien créer une nouvelle `bobcase` une fois sur k. — 2 pts

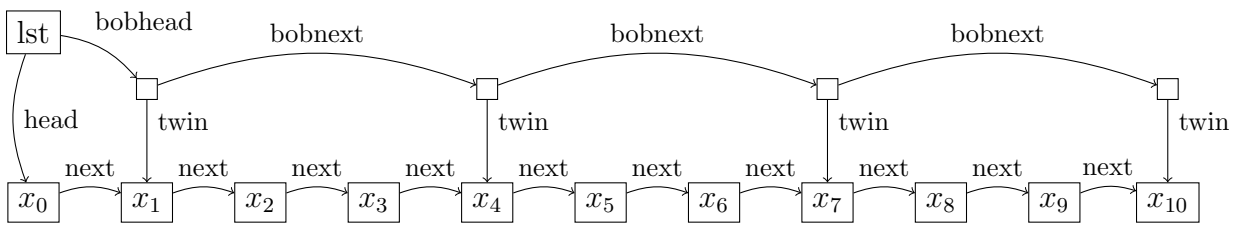

FIGURE 1 – Exemple d'une structure de boblist à 11 éléments, avec $k = 3$.

3. Quel problème Bob aurait-il rencontré au moment de l'ajout d'un nouvel élément dans une boblist s'il avait créé une case jumelle pour les cases à distance $i * k$ du début de la liste plutôt que de la fin ? 1 pt

Bob cherche maintenant à trouver la bonne valeur à fixer pour k. En analysant la complexité d'accès dans le pire cas et dans le but de la minimiser, il décide de faire en sorte que $k = \Theta(\sqrt{n})$ quand sa boblist `lst` est de taille n. Il initialise donc la valeur de `lst.k` à 1, et il devra la mettre à jour régulièrement en fonction de `lst.n`, ce qui l'obligera à changer les pointeurs en conséquence !

4. Si après avoir ajouté un élément à `lst` on obtient $\texttt{lst.n} \geq 4 \times \texttt{lst.k}^2$, alors on double la valeur de `lst.k` et on met à jour les pointeurs `bobnext` de chaque `bobcase` de façon adéquate. Écrire un algorithme `update(lst)` qui fait ce test et la mise à jour subséquente le cas échéant. Quelle est sa complexité ? 3 pts

5. Montrer que grâce à cette mise à jour, on a toujours $\frac{\sqrt{\texttt{lst.n}}}{2} \leq \texttt{lst.k} \leq \sqrt{\texttt{lst.n}}$ (si $\texttt{lst.n} \geq 1$). 2 pts

   Conseil *: Faire une preuve par récurrence sur n.*

6. À cause de la nécessité de ces mises à jour de k, la complexité de l'insertion d'un élément n'est plus systématiquement constante. Montrer cependant que ces mises à jour arrivent suffisamment rarement pour que le coût amorti de l'insertion reste $O(1)$ (i.e. la complexité totale de l'insertion de N éléments dans une boblist initialement vide est $O(N)$). 2 pts

Bob a réussi à mettre au point une structure où l'insertion est en temps amorti constant, et l'accès à n'importe quel élément en temps $O(\sqrt{n})$. Il est satisfait, mais il sait qu'il pourrait faire encore mieux !

7. Joe, un camarade de classe de Bob, lui fait remarquer que la structure de tableau dynamique permettait déjà de faire l'insertion d'un élément en temps amorti $O(1)$, et a une complexité d'accès bien meilleure, en temps $O(1)$. Bob, vexé, lui rétorque que sa structure est quand même meilleure pour gérer l'insertion d'un élément en une position $i \in [0, n-1]$ fixée. Écrire un algorithme `insertAt(lst,`x`,`i`)` permettant d'insérer une nouvelle `case` de valeur x à la position i dans une `boblist lst`. Justifier que sa complexité est meilleure que l'opération similaire dans un tableau dynamique. 4 pts

8. Maintenant convaincu que sa structure de `boblist` est pertinente, Bob a l'ambition de l'enrichir en utilisant une troisième liste, et ainsi obtenir une complexité $O(\sqrt[3]{n})$ dans le pire cas pour accéder à n'importe quelle position de sa `boblist`, tout en gardant une complexité amortie $O(1)$ pour l'insertion en tête de `boblist`. Donner la définition de cette structure enrichie, et donner la nouvelle version de `getElement` s'appuyant sur cette structure. 3 pts