TD 6

Partie A : tableaux statiques

Exercice 1

On considère un ensemble de k données stockés dans un tableau de données statique T de n cases (avec $0 \leq k \leq n$).

Remarques :

- Les cases du tableau sont numérotées de `0` à `n-1`
- Les données sont de type quelconque mais chaque case ne peut contenir qu'une donnée
- Si `i` est un indice de case, `T[i]` désigne le contenu de la case
- n est fixé mais k varie en fonction du nombre de données stockées

1. Le stockage est dense, autrement dit: les données sont stockés dans les k premières cases du tableau. Ainsi, les cases de `0` à `k-1` sont occupées et l'indice k désigne la première case libre.

- Ecrire un algorithme permettant d'insérer une nouvelle donnée `t` dans le tableau T
- Ecrire un algorithme de *recherche* qui prend en argument une donnée `t` et retourne :
  - le numéro de toutes les cases contenant `t` si `t` est présent dans le tableau
  - une liste vide sinon ("donnée absente")
- Ecrire un algorithme de *suppression* prend en argument une donnée `t` et :
  - supprime toutes les occurrences de `t` du tableau si `t` est présent dans le tableau
  - Ne fait rien sinon
- Donner la complexité de ces algorithmes.

2. On suppose maintenant qu'il n'y a pas de doublons dans le tableau, autrement dit $\forall$ `i`,`j` < k, si `i` ≠ `j` alors `T[i]` ≠ `T[j]`. Réécrire les algorithmes de recherche, d'insertion et de suppression et donner leur complexité.

3. On suppose maintenant qu'il existe un ordre ≺ sur les données. $\forall$ `i`,`j` < k, si `i` < `j` alors `T[i]` ≺ `T[j]`. Réécrire les algorithmes de recherche, d'insertion et de suppression et donner leur complexité.

4. Que faire quand le tableau est plein?

Exercice 2 (*)

On appelle *liste* une structure abstraite ordonnée telle que l'on puisse accéder de manière directe à l'élément `i` et à laquelle on puisse ajouter (et supprimer) autant d'éléments que l'on souhaite. Une caractéristique importante de cette structure est son nombre d'éléments k.

Une implémentation des listes peut être effectuée comme suit:

- On commence par créer un tableau de taille n = `1`, le nombre initial d'éléments étant k = `0`
- A chaque ajout d'élément:
  - si `k < n`,

        - ajouter l'élément à la position k
        - k ← `k + 1`
    - sinon :
        - allouer un tableau à `2 * n` éléments et n ← `n * 2`
        - copier les k premiers éléments du tableau initial dans le nouveau tableau & supprimer le tableau initial.
        - ajouter l'élément à la position k
        - k ← `k + 1`

Montrez que la complexité de l'ajout de k éléments à la fin d'une liste originellement vide est O(k).

Partie B : Tables de hachage

Soit U un “univers” dont les éléments sont appelés *clés*. Soit E un ensemble de clés. On suppose que l'on a une fonction h:U –> `{0`, … `n-1}`, dite *fonction de hachage* (ou *hashcode*). Une *table de hachage* est un tableau T de taille n tel que `T[i]` est une liste contenant les éléments x de E tels que `h(x)=i`. Si deux éléments de E ont le même hashcode, on dit qu'il y a *collision*.

Exercice 1

Donnez des algorithmes pour rechercher, insérer & supprimer un élément dans une table de hachage. Donnez leur complexité dans le cas le meilleur, le pire & en moyenne.

(**) Déduisez-en la valeur optimale (en ordre de grandeur) de n en fonction du nombre d'éléments stockés k, ainsi qu'une contrainte sur la fonction de hachage.

Exercice 2

Il arrive souvent que l'on ne sache pas à l'avance combien d'éléments contient E & que l'on mette les éléments de E dans T l'un après l'autre sans savoir quand on s'arrêtera. Donnez une ``politique" efficace de gestion de la taille de T.

Exercice 3 (*)

Soit S un ensemble de nombres à trier, on répartit S en une table de hachage tel que la fonction de hachage soit croissante (x≤ y ⇒ h(x) ≤ h(y)). On trie chaque paquet, puis on concatène. On appelle ce tri le *tri par paquets*.

- Donnez une fonction de hachage simple & croissante.
- Quelle est la complexité de cet algorithme dans le cas le meilleur, le pire & en moyenne.

Partie C : dictionnaires

Un *dictionnaire* est une structure de données (python) qui se présente ainsi:

```
D = {clé_1:valeur_1, clé_2:valeur_2, ..., clé_n:valeur_n}
```

Les clés pouvant être de (presque) n'importe que type (& pas seulement l'ensemble `{0,…,n-1}` comme avec une liste).

- On accède à `valeur_i`, la valeur associée à `clé_i` par `D[clé_i]`.
- L'opération `D[clé_p]` ← `valeur_p`,
  - si `clé_p` n'est pas une clé de `D`, ajoute cette nouvelle clé à `D` & lui associe la valeur `valeur_p`
  - si `clé_p` est déjà une clé de `D`, elle change la valeur qui lui est associée en `valeur_p`.

Exercice 1

Donnez une implémentation efficace des dictionnaires. Quelle est alors la complexité (dans le cas le meilleur, le pire & en moyenne) des fonctions de base (recherche, ajout d'un élément,…) sur un dictionnaire.

Exercice 2

Utilisez un dictionnaire pour écrire un algorithme qui compte le nombre d'occurrences de chaque mot d'un texte.

Exercice 3

Utilisez un dictionnaire pour écrire un algorithme qui supprime les doublons d'une liste. Donnez sa complexité (dans le cas le pire, le meilleur & en moyenne).