EA4 – Éléments d'algorithmique
TD nº 12 : révisions

Exercice 1 : densité (d'après examen 2021)

Dans cet exercice, on considère des tableaux de nombres (non nécessairement entiers), supposés tous distincts pour simplifier.
On définit l'élément *le plus isolé* d'un tel tableau `T` comme celui qui maximise la distance à son plus proche voisin. Par exemple, dans `T = [ 5.4, 3.2, 4.7, 1.5, 0.9, 1.8 ]`, l'élément le plus isolé est `3.2`, à distance `1.4` de son plus proche voisin, `1.8`.

1. Décrire un algorithme naïf `plus_isole_naif(T)` permettant de résoudre ce problème sans modifier le tableau `T` et avec mémoire auxiliaire constante.

2. Proposer un algorithme `plus_isole_efficace(T)` plus efficace que l'algorithme naïf. Justifier sa correction et sa complexité (au pire et en moyenne).

La distance `r` entre l'élément le plus isolé et son plus proche voisin permet de définir une mesure de *densité* : la densité autour d'un élément `x` est le nombre d'éléments de `T` à distance strictement inférieur à `r` de `x`. Elle vaut donc 1 pour l'élément le plus isolé de `T` et au moins 2 (voire beaucoup plus) pour tous les autres éléments de `T`.

3. Définir un algorithme `densite_tab_trie(T, x, r)` le plus efficace possible pour déterminer la densité de `T` autour de l'élément `x` en supposant que `T` est trié. Quelle est sa complexité ?

4. Supposons qu'on souhaite calculer les densités autour d'un nombre m d'éléments d'un même tableau `T` *initialement non trié*. Pour quels ordres de grandeur de m est-il judicieux de trier `T` ?

Exercice 2 : B-arbres (d'après examen 2017)

Les *B-arbres d'ordre* p constituent une variante des arbres binaires de recherche, utilisée notamment pour les systèmes de gestion de fichiers. Les différences majeures sont :

- chaque nœud ou feuille contient au plus $2p$ clés ;
- chaque nœud ou feuille (***sauf la racine***) contient au moins p clés ;
- la racine d'un B-arbre non vide contient au moins une clé ;
- un nœud d'arité $k+1$ contient exactement k clés ;
- ***toutes les feuilles ont la même profondeur.***

La propriété d'ordre des ABR s'étend quant à elle simplement aux nœuds d'arité $k+1$: si un nœud contient les clés $c_0 < c_1 < \cdots < c_{k-1}$ et possède les sous-arbres A_0, A_1... A_k, tous les éléments de A_i sont supérieurs à c_{i-1} (si $i > 0$) et inférieurs à c_i (si $i < k$).

1. a. Quelle est la seule forme possible pour un B-arbre d'ordre p contenant au plus $2p$ clés ? Et exactement $2p+1$ clés ?

b. Quelles sont les deux formes possibles pour un B-arbre d'ordre 1 et de hauteur 1 ? Combien chacune d'elles peut-elle contenir de clés ?

c. Décrire toutes les formes possibles pour un B-arbre d'ordre p et de hauteur 1. Combien de clés un tel B-arbre peut-il contenir ?

d. Quelles sont les hauteurs minimale et maximale d'un B-arbre d'ordre 1 contenant 15 clés ? Donner un exemple de chaque hauteur possible (avec comme clés les entiers de 1 à 15).

2. Donner une minoration du nombre de clés à profondeur k, pour $k > 0$ (en fonction de p). En déduire que la hauteur d'un B-arbre contenant n clés est en $\Theta(\log n)$ dans tous les cas.

Par souci de simplification, on suppose toutes les clés distinctes, et on considère que chaque `sommet` contient :

- un champ booléen `feuille` indiquant s'il s'agit d'une feuille ou non,
- un champ entier `taille` compris entre p et $2p$ indiquant le nombre de clés qu'il contient,
- un tableau `cles` de longueur $2p$, trié, contenant les clés [1],
- un tableau `fils` de longueur $2p + 1$ contenant les fils[1], dans l'ordre,

pour lesquels on dispose de tous les accesseurs nécessaires – par exemple, `getTaille(sommet)`, `getCles(sommet)`, `getCle(i, sommet)`...

3. Décrire un algorithme `minimum(racine)` qui retourne le plus petit élément du B-arbre dont `racine` est la racine. Quelle est sa complexité ?

4. Décrire un algorithme `listeTriee(racine)` retournant la liste triée de tous les éléments du B-arbre dont `racine` est la racine. Quelle est sa complexité ?

5. Décrire un algorithme `estUnBArbre(racine)` retournant `True` si l'arbre dont `racine` est la racine est un B-arbre valide, et `False` sinon. Quelle est sa complexité ?

6. Décrire un algorithme `appartient(c, sommet)` *le plus efficace possible* retournant
- `True` si `sommet` contient la clé `c`,
- `False` si `sommet` est une feuille ne contenant pas `c`,
- et l'unique sous-arbre de `sommet` susceptible de contenir `c` sinon.

Quelle est sa complexité (en fonction de p, qui a vocation à être grand) ?

7. En déduire un algorithme `cherche(c, racine)` *le plus efficace possible* retournant le nœud du B-arbre de racine `racine` contenant `c`, s'il en existe, et `False` sinon.

8. Quelle est la complexité de `cherche(c, racine)`, en fonction de p et du nombre n de clés stockées dans le B-arbre de racine `racine` ?

L'ajout de nouveaux éléments est plus complexe, du fait de la contrainte sur la profondeur des feuilles : comme dans un ABR, on cherche à ajouter l'élément dans une feuille, mais s'il est nécessaire d'en créer une nouvelle, elle doit être au même niveau que les précédentes – ce qui peut avoir des répercussions sur son père, voire toute sa lignée ancestrale. S'il faut finalement augmenter la hauteur de l'arbre, cela devra se faire au niveau de la racine.

9. Comment créer un B-arbre d'ordre 2 en ajoutant successivement les clés 1, 2, 3, 4, 5 ? Et un B-arbre d'ordre 1 ? Poursuivre dans chacun des deux cas avec l'insertion de 6, 7, 8.

10. Décrire l'ajout d'une nouvelle clé dans une feuille non saturée.

11. ($*$) Proposer un algorithme pour le cas général. Quelle est sa complexité ?

1. et `None` dans les cases inutilisées