Analyse d'algorithmes et programmation - 1^{er} contôle continu

Enseignants: Christina Boura, Luca De Feo

19 février 2016

Durée du contrôle : 1h30.

Les documents ainsi que l'accès aux programmes élaborés pendant les séances de TP sont autorisés.

Bonne chance !

1 Questions

Exercice 1 Soient $f(n)$ et $g(n)$ deux fonctions dont le domaine de définition est l'ensemble des entiers naturels $\mathbb{N}$. On suppose que $f(n)$ et $g(n)$ sont asymptotiquement non négatives. En utilisant la définition de base de la notation Θ, démontrer que

$$\max(f(n), g(n)) = \Theta(f(n) + g(n)).$$

Exercice 2 Est-ce que $2^{n+1} = \mathcal{O}(2^n)$? Est-ce que $2^{2n} = \mathcal{O}(2^n)$? Justifier votre réponse.

Exercice 3 Un *vecteur de bits* est tout simplement un tableau de bits (0 et 1). Un vecteur de bits de longueur m prend beaucoup moins d'espace à stocker qu'un tableau de m pointeurs. Décrire comment on pourrait utiliser un vecteur de bits pour représenter un ensemble dynamique de m éléments distincts sans données satellites. Les opérations de dictionnaire (recherche, insertion et suppression d'un élément) devront s'exécuter dans un temps $\mathcal{O}(1)$.

Exercice 4 Montrer comment on réalise l'insertion des clés 5, 28, 19, 15, 20, 33, 12, 17, 10 dans une table de hachage où les collisions sont résolues par chaînage. On suppose que la table contient 9 cases et que la fonction de hachage est $h(k) = k \mod 9$.

Exercice 5 Montrer que, si un nœud d'un arbre binaire de recherche a deux enfants, alors son successeur n'a pas d'enfant de gauche et son prédécesseur n'a pas d'enfant de droite. (Le *successeur* d'un nœud x est le nœud possédant la plus petite clé supérieure à clé$[x]$. De la même manière, le *prédécesseur* d'un nœud x est le nœud possédant la plus grande clé inférieure à clé$[x]$.

Exercice 6 On peut trier un ensemble donné de n nombres en commençant par construire un arbre binaire de recherche contenant ces nombres (en répétant l'algorithme d'insertion pour insérer les nombres un à un), puis en imprimant les nombres via un parcours infixe de l'arbre. Quels sont les temps d'exécution de cet algorithme de tri, dans le pire et dans le meilleur des cas ?

2 Programmation

1. Générez une liste de 100 entiers aléatoires dans l'intervalle $[0, 1000]$ et affichez-là. Pour générer un entier aléatoire dans l'intervalle $[a, b]$ vous pouvez utiliser l'instruction `randint(a,b)`. Pour l'utiliser, insérez dans votre programme la ligne `from random import randint`.
2. Créez une classe `Noeud` :

```
class Noeud :

     def __init__(self,entier) :
        self.gauche = None
        self.droite = None
        self.cle = entier

     def inserer(self, entier) :
        # A compléter

     def parcoursInfixe(self) :
        # A compléter

      def profondeurMaximale(self) :
        # A compléter
```

3. Complétez la méthode `inserer` qui prend en entrée un entier et qui insère dans l'arbre le noeud ayant comme clé cet entier.
4. Complétez la méthode `parcoursInfixe` pour afficher les clés stockées dans l'arbre en suivant un parcours infixe.
5. Triez votre liste en insérant d'abord tous ses éléments dans l'arbre et en les affichant ensuite selon un parcours infixe. **Astuce :** Afin de ne pas commencer avec un arbre vide, vous pouvez avant le parcours de votre liste écrire `noeud = Noeud(L[0])`, où `L` est votre liste des entiers aléatoires.
6. Complétez la méthode `profondeurMaximale` qui affiche la hauteur de l'arbre, c'est-à-dire la distance maximale d'une feuille de l'arbre de la racine.