Algorithmique et structures de données : Mission 3

Groupe 1.2: Ivan Ahad - Jérôme Bertaux - Rodolphe Cambier
Baptiste Degryse - Wojciech Grynczel - Charles Jaquet

24 octobre 2014

Question 1 (Charles Jacquet)

- **Les clés doivent-elles automatiquement être des nombres**
  Non, elles peuvent être n'importe quoi tant que c'est comparable. Par exemple, ça pourrait être des String classé de manière alphabétique.
- **Enumérer en ordre croissant toute les clés mémorisées**
  il suffit d'utiliser une fonction récursive, qui va se réappeller à chaquer élément de telle sorte que : String s = recursiveFunction(tree);

  avec comme pseudo code

  ```
  public String recursiveFunction(BinaryTree tree){
  if (tree.left == null && tree.right == null){
  return tree.getElem();
  }
  else if(tree.left == null){
  return recursiveFunction(tree.right);
  }
  else if(tree.right == null){
  return recursiveFunction(tree.left);
  }
  else{
  return recursiveFunction(tree.left) + recursiveFunction(tree.rigth);
  }
  }
  ```

  La complexité de cette méthode est en O(h) avec h la hauteur du root.
- **Dans le cas où une clé est mémorisée deux fois**
  Lors de la deuxième mémorisation, dans le livre il est marqué qu'elle remplace la première. Il n'y a donc pas de relation père-fils.

Question 2

Question 3 (Bertaux Jérôme)

```
/*
* PRE : t est un arbre trié de manière croissante depuis le sous arbre de gauche vers le sous arbre d
* POST : l'entrée possédant la plus petite clé ou null si l'arbre est vide.
*
* La complexité est de l'ordre de O(h)
*/
public Entry firstEntry(Tree t){
if(t.isEmpty()){
return null;
}else{
```

```
Tree tmp = t;
while(t2.hasLeft()){
t2 = t2.getLeft();
}
return t2.getValue();
}
}

/*
* PRE : t est un arbre trié de manière croissante depuis le sous arbre de gauche vers le sous arbre d
k une clé
* POST : l'entrée possédant une clé plus grande que k ou null si elle n'existe pas.
*
* La complexité est de l'ordre de O(h)
*/
public Entry higherEntry(Tree t, int k){
if(!t.isEmpty()){
if(t.getValue().getKey() > k){
return t.getValue();
}else if(t.hasRight()){
higherEntry(t.getRight(), k);
}
}else{
return null;
}
}
```

Question 4

Question 5

Question 6

Question 7

Question 8 (Baptiste Degryse) La propriété la moins évidente à vérifier est l'équilibre de l'arbre. Puisqu'il y a 1000 éléments, la profondeur de l'arbre doit être de maximum $log_2(1000) = 9.9 \simeq 10$. Donc, pour la racine, il doit y avoir entre 512 et 488 clés dans chaque sous-arbre si toutes les clés sont utilisées. Sinon, il faut avoir un minimum de 2^{n-1} où n est la hauteur visible lors de la recherche.

2, 252, 401, 398, 330, 344, 397, 363 : Impossible, l'arbre n'est pas équilibré (donc pas AVL) car il y a maximum 1 enfant à gauche de la racine. (le chiffre 1)

924, 220, 911, 244, 898, 258, 362, 363 : $2^{7-1} = 64$. Il n'y a donc pas de problème pour celui-ci, mais il y a une grande densité de présence de clé pour les clés élevées.

925, 202, 911, 240, 912, 245, 363 : idem

2, 399, 387, 219, 266, 382, 381, 278, 363 : Impossible, l'arbre n'est pas équilibré (donc pas AVL) car il y a maximum 1 enfant à gauche de la racine. (le chiffre 1)

935, 278, 347, 621, 299, 392, 358, 363 : il faut 64 éléments dans l'autre sous arbre, ce qui implique que toutes les clés sauf une doivent être présentes au dessus de 934 pour que l'arbre soit valide.