IFT2015 A12 — Examen Final

Miklós Csűrös

11 janvier 2013

Aucune documentation n'est permise. L'examen vaut 150 points, et vous pouvez avoir jusqu'à 25 points de boni additionnels. Vous pouvez décrire vos algorithmes en pseudocode ou en Java(-esque).

► **Répondez à toutes les questions dans les cahiers d'examen.**

F0 Votre nom (1 point)

► Écrivez votre nom et code permanent sur tous les cahiers soumis.

F1 Table de symboles (39 points)

(a) 3 structures, 3 opérations, 3 performances (27 points) On a vu plusieurs structures de données qui peuvent servir à implémenter le type abstrait de la table de symboles. L'efficacité des implémentations n'est pas la même : ici vous devez comparer le temps de calcul pour trois opérations fondamentales : insertion, recherche fructueuse, et recherche infructueuse. ► Donnez le temps de calcul des trois opérations avec les trois structures de données suivantes : un tableau d'éléments triés (avec une capacité assez grande pour l'insertion), un arbre rouge-et-noir, et un tableau de hachage avec sondage linéaire, dont la facteur de remplissage $\alpha < 0.9$. Spécifiez le temps de calcul comme une fonction du nombre des éléments n, en utilisant la notation asymptotique, dans trois cas : le pire cas, le meilleur cas, et en moyenne. Il ne faut pas justifier vos réponses.

(b) Justifications (12 points) ► Élaborez *3* de vos réponses (votre choix lesquels parmi les 27) en **(a)** : expliquez comment la structure assure un tel temps de calcul.

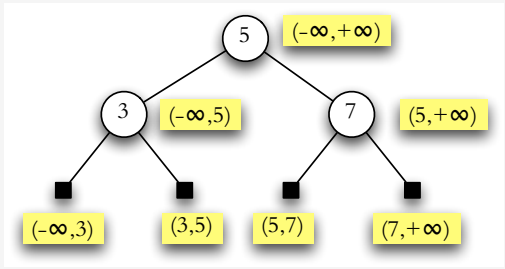
F2 Tris (20 points)

► Pour chacun des tris suivants, donnez le temps de calcul en meilleur, moyen et pire cas, ainsi que l'espace de travail utilisé au pire : tri rapide, tri par tas, tri par fusion, tri par insertion, tri par sélection. Donnez vos réponses en notation asymptotique pour un tableau de taille n. (Notez que l'espace de travail inclut toute les variables locales à part du tableau à l'entrée, et dépend de la profondeur de la pile d'exécution quand le tri utilise de la récursion.) Il n'est pas nécessaire de justifier vos réponses.

F3 Types abstraits (20 points)

► Décrivez les opérations principales pour les types abstraits suivants, sans discuter l'implantation : (1) pile, (2) file FIFO (queue), (3) file de priorité, (4) table de symboles.

F4 Gamme de clés (25 points)



La **gamme** de chaque nœud x (interne ou externe) dans un rabre binaire de recherche est une paire $\big(\alpha(x), \beta(x)\big)$ vérifiant la propriété suivante : toutes clés possibles (insérées ou à insérer) dans le sous-arbre enraciné à x sont supérieures à $\alpha(x)$ et inférieures à $\beta(x)$. Notez qu'il est permis d'avoir $\alpha(x) = -\infty$ et/ou $\beta(x) = \infty$: la gamme de la racine est toujours $(-\infty, \infty)$.

a. Définition (10 points). ▶ Donnez une définition récursive pour la gamme d'un nœud interne.

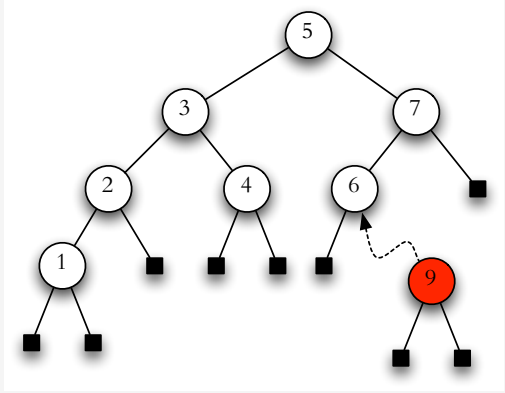
b. Algorithme (15 points). ▶ Donnez un algorithme récursif qui affiche la gamme associé à chaque nœud dans un ABR. L'algorithme doit prendre $O(n)$ temps sur un arbre à n nœuds : arguez que c'est le cas avec votre solution.

F5 Est-ce qu'une brébis galeuse gâte le troupeau ? (20 points)

On a un arbre binaire de recherche ordinaire, construit par une série d'insertions à partir d'un arbre vide. Plus précisement on exécute root ← INSERT(root, v_i) pour une séquence de clés différentes $v_1, v_2, \ldots, v_n$:

```
INSERT(x, v)        // insère la clé v dans le sous-arbre de x et retourne la nouvelle racine
I1  if x est externe then y ← nouveau nœud avec clé v ; return y
I2  if v < x.key
I3  then x.left ← INSERT(x.left, v)
I4  else x.right ← INSERT(x.right, v)
I5  return x
```

Chaque nœud contient les variables x.key, x.left et x.right pour clé la clé, l'enfant gauche et l'enfant droit (pas de variable parent). Un nœud x est dit *malplacé* s'il est dans le sous-arbre gauche d'un ancêtre y avec x.key $> y$.key, ou s'il est dans le sous-arbre droit d'un ancêtre y avec x.key $< y$.key.



Supposons que lors de l'insertion «voyou» d'une clé quelconque, le nouvel nœud a été crée à un mauvais emplacement (mais toujours en remplaçant un nœud externe). À ce point, il existe donc un seul nœud malplacé dans l'arbre. On se demande si les insertions ensuite mènent à de nouveaux nœuds malplacés. Professeur Tournesol soutient que le théorème suivant applique.

Théorème F5.1. *Si on performe un nombre arbitraire d'insertions correctes après une seule insertion voyou d'une clé quelconque, le nouveau nœud reste le seul nœud malplacé, et un de ses enfants reste toujours externe. (On ne supprime aucun noeud.)*

▶ Démontrez le théorème de prof. Tournesol, ou montrez un contre-exemple.

F6 Comment battre les cartes ? (25+25 points)

Le but de battre les cartes est d'obtenir un ordre aléatoire dans le paquet. Mathématiquement, on veut une permutation aléatoire uniformement distribuée. L'algorithme suivant construit une telle permutation.

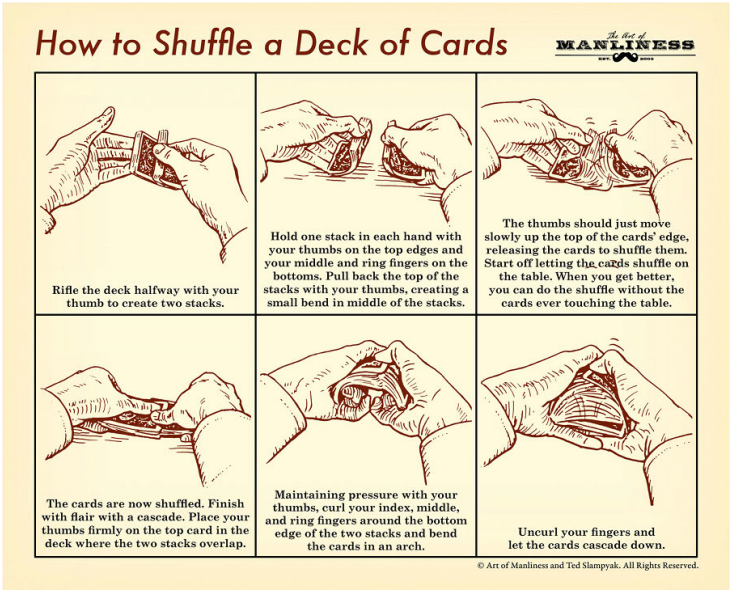
```
PERM(n)                                   // (construit une permutation aléatoire π[0..n − 1])
P1  initialiser for i ← 0, 1, 2, ..., n − 1 do π[i] ← i
P2  for i ← 0, 1, ..., n − 2 do
P3      j ← Random(i, i + 1, ..., n − 1)  // (une des valeurs i, i + 1, ..., n tirée au hasard)
P4      échanger π[i] ↔ π[j]
P5  return π
```

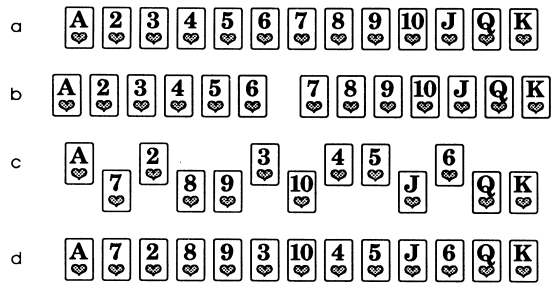
En Ligne P3, on se sert d'un générateur de nombres pseudoaléatoires pour choisir l'indice j uniformement distribué parmi $i, i+1, \ldots, n-1$.

a. Permutation d'une liste chaînée (25 points). ►Adaptez la logique de PERM, pour calculer la permutation aléatoire d'une liste simplement chaînée. Plus précisement, donnez le pseudocode pour LISTPERM(x, n) qui retourne la tête d'une liste avec n nœuds (par exemple, n cartes d'un jeu) après une permutation aléatoire de distribution uniforme. L'argument x est la tête de la liste initiale avec n nœuds. ►Analysez le temps de calcul de votre algorithme (Random prend $O(1)$).

♡**b. Battage (25 points boni).** ► Proposez un algorithme qui implante le battage par feuilleter (le «riffle shuffle», illustré ci-dessous) à l'aide de listes simplement chaînées. ► Analysez le temps de calcul de l'algorithme*.

Indice: *Remplacez la comparaison par choix aléatoire dans le tri par fusion.*





À l'entrée (a), on a une liste $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ et un argument $k \in \{0, 1, \ldots, n\}$. L'algorithme (b) découpe la liste en $A = (x_1, \ldots, x_k)$ et $B = (x_{k+1}, \ldots, x_n)$ (les listes peuvent être vide), (c) entrelace les deux listes au hasard, (d) et ainsi construit la fusion aléatoire des deux listes, où l'ordre original entre les éléments de la même sous-liste reste le même.

BONNE CHANCE !

* Par rapport à LISTPERM, il suffit de refaire le battage $O(\log n)$ fois pour convergence à la distribution uniforme [Dave Bayer et Persi Diaconis, 1992].

English translation

No documentation is allowed. The examen is worth 150 points, and you can collect an additional 25 bonus points. Describe your algorithms in pseudocode or Java(-esque).

Answer each question in the exam booklet.

E0 Your name (1 point)

► Write your name and *code permanent* on each booklet that you submit.

E1 Symbol table (39 points)

(a) 3 structures, 3 operations, 3 performance guarantees (27 points) We have seen a number of data structures that can be used to implement the abstract data type of symbol table. Not all implementations have the same performance. You need to compare the running times for three fundamental operations : insertion, successful search and unsuccessful search. ► Give the running times for the three operations in the three following data structures : sorted table (with enough capacity for the insertion), red-black tree, and hash table with linear probing and a load factor of $\alpha < 0.9$. Give the worst-case, best-case and average-case running times in asymptotic notation as a function of the number n of elements. (That is, 27 statements about time complexity.) You do not need to justify your answers.

(b) Justifications (12 points) ► Develop *3* of your answers (of your choice among the 27) from **(a)**. Explain how the structure allows for such performance.

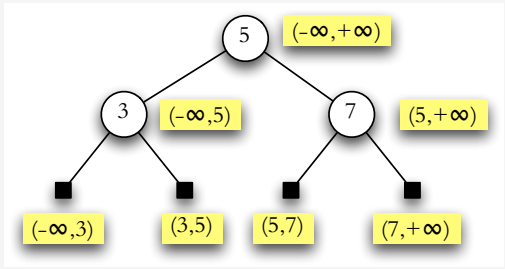
E2 Sorting (20 points)

► Give the running time in the best, worst, and average case, as well as the worst-case work space requirements for the following sorting algorithms : quicksort, heapsort, mergesort, insertion sort, and selection sort. Give your answers in asymptotic notation for a table of n elements. (Note that the work space includes all allocated local variables aside from the input table, and depends on the depth of the call stack when recursion is used.) You do not need to justify your answers.

E3 Abstract types (20 points)

► Describe the fundamental operations for each of the following abstract data types, without discussing implementation : (1) stack, (2) queue, (3) priority queue, (4) symbol table.

E4 Key range (25 points)



Define the key **range** $\big(\alpha(x), \beta(x)\big)$ for every node x (internal or external) in a binary search tree as the set of possible keys that are or would be inserted into the subtree rooted at x. Note that one can have $\alpha(x) = -\infty$ and/or $\beta(x) = \infty$: the root's range is always $(-\infty, \infty)$.

a. Definition (10 points). ► Give a recursive definition for the key range of an internal node x.

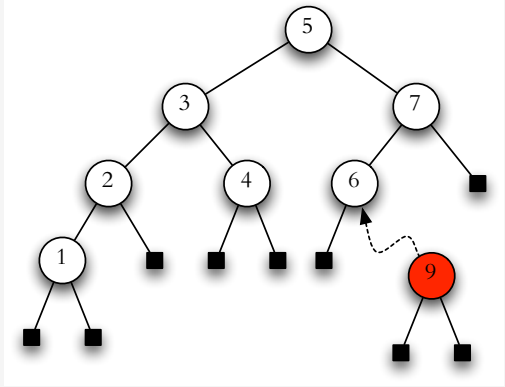
b. Algorithm (15 points). ► Give a recursive algorithm that lists the key ranges for all nodes in a BST. The algorithm mus take $O(n)$ time on a tree with n nodes : argue that such is the case for your solution.

E5 One bad apple (20 points)

Suppose that we have an ordinary binary search tree, built through a series of insertions into en empty tree. Specifically, the tree is built by executing root $\leftarrow$ INSERT(root, v_i) for a sequence of distinct keys $v_1, v_2, \ldots, v_n$:

```
INSERT(x, v)          // inserts the key v into the subtree of x and returns the new root
I1  if x is external then y ← new node with key v ; return y
I2  if v < x.key
I3  then x.left ← INSERT(x.left, v)
I4  else x.right ← INSERT(x.right, v)
I5  return x
```

Every internal node x contains the variables x.key, x.left and x.right for key, left child and right child, but there are no parent variables. A node x is said to be **misplaced** if and only if it is in the left subtree of an ancestor y with x.key $> y$.key, or it it is in the right subtree of an ancestor y with x.key $< y$.key.



Suppose that there is a "rogue" insertion where a new node is created by replacing the wrong external node. At this point, there is exactly one misplaced node in the tree. Will new insertions create more misplaced nodes ? Professor Calculus thinks not, and postulates the following theorem :

Théorème E5.1. *If a single rogue insertion is followed by an arbitrary number of correctly executed insertions and no deletions, exactly one node remains misplaced throughout the procedure. Moreover, at least one of the misplaced node's children stays always external.*

► Prove Prof. Calculus' theorem, or show a counterexample.

E6 How to shuffle cards? (25+25 points)

The goal of shuffling is to establish a random order in a deck of cards. Mathematically, one wants a random, uniformly distributed permutation. The following algorithm computes a uniformly random permutation.

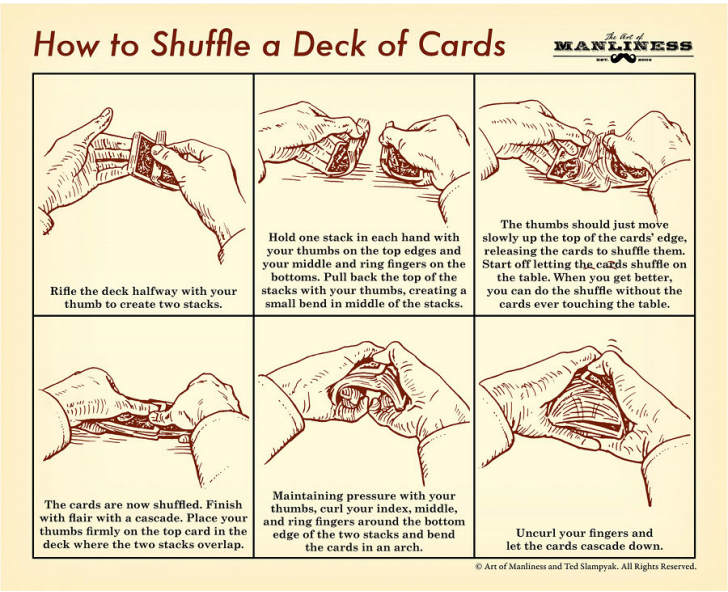
```
PERM(n)                                          // (establishes a random permutation π[0..n − 1])
P1  initialization : for i ← 0, 1, 2, ..., n − 1 do π[i] ← i
P2  for i ← 0, 1, ..., n − 2 do
P3      j ← Random(i, i + 1, ..., n − 1)         // (one of i, i + 1, ..., n picked randomly)
P4      exchange π[i] ↔ π[j]
P5  return π
```

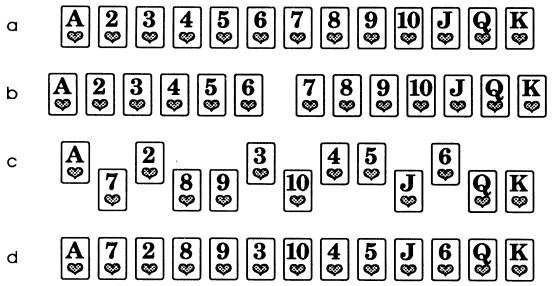
In Line P3, we use a pseudorandom number generator to pick an index j uniformly distributed between $i, i+1, \ldots, n-1$.

a. Permutation of a linked list (25 points). ▶Adapt the logic of PERM to randomly permute a linked list. More precisely, give the pseudocode for an algorithm LISTPERM(x, n) that returns the head of a list with n nodes (e.g., n cards) after a random, uniformly distributed permutation. The argument x is the head of a list with n nodes. Every node contains the variables x.next and x.val (next node and value). ▶Analyze the running time of your algorithm (Random takes $O(1)$).

♡**b. Card shuffling (25 points boni).** ▶ Give an algorithm that implements the "riffle shuffle " (see ilustration below) using singly linked lists. ▶ Analyze your algorithm's running time†.

Hint: *Replace the comparison with random choice in mergesort.*





The input (a) is a list $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of cards and an argument $k \in \{0, 1, \ldots, n\}$. The algorithm (b) cuts the list into $A = (x_1, \ldots, x_k)$ and $B = (x_{k+1}, \ldots, x_n)$ (lists may be empty), (c) interlaces the two lists (d) by randomly merging the two lists. The original order between elements of the same sublist stays the same.

GOOD LUCK!

† For comparison with LISTPERM : it is enough to shuffle $O(\log n)$ times to converge to the uniform distribution [Dave Bayer and Persi Diaconis, 1992].