Questions récurrentes

January 6, 2020

1 Types abstraits et Structure de données

1.1 Qu'est ce qu'un variant ? A quoi sert-il ? Quelles sont les règles de preuves à employer ?

Le variant, aussi appelé "fonction de terminaison", prouve la terminaison d'une boucle. Il s'agit d'une expression donnant une borne supérieure sur le nombre de fois qu'il faut encore passer dans la boucle. Ce variant doit diminuer à chaque tour de boucle :

$$sp(S, I \wedge B \wedge V = v_0) \Rightarrow V < v_0 \tag{1}$$

Si ce variant est égale à 0, alors nous avons $\neg B$ et la boucle se termine.

1.2 Qu'est-ce qu'un invariant de boucle ? Quand faut-il le prouver ou le supposer ?

Un invariant de boucle désigne une propriété vraie en début de boucle et en fin de boucle. Il est désigné par la lettre I. Il faut le prouver afin de vérifier l'exactitude de la boucle, pour cela, il faut prouver :

1. de l'implication $P' \Rightarrow I$ (où P' est la précondition avant l'entrée dans la boucle),
2. de l'implication $sp(S, I \wedge B) \Rightarrow I$ (un passage dans la boucle rétabli l'invariant)
3. de l'implication $I \wedge \neg B \Rightarrow Q'$ (pu Q' est la postcondition à la sortie de la boucle, autrement dit, ce que la boucle est censée faire)

1.3 Qu'est-ce qu'un invariant de données ? Quand faut-il le prouver ou le supposer ?

Un invariant de données désigne un ensemble de propriété sur les données concrètes qui est vrai à tout moment pendant l'existence des dites données. L'invariant est supposé vrai avant tout appel à une fonction et doit être prouvé après l'appel de cette fonction.

1.4 Définissez le type abstrait dictionnaire

Un dictionnaire S est un ensemble dynamique d'éléments $x \in U$ avec des clés $k \in \varepsilon$ comparables: on peut supposer qu'il existe un ordre total sur les clés. Un description fonctionnelle de ses opérations est :

- search: $S \times \varepsilon \mapsto U$,
- insert: $S \times U \times \varepsilon \mapsto void$
- delete: $S \times U \mapsto void$

Le dictionnaire est aussi un object conteneur. Il n'a quand à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objects contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des **clés** qui peuvent être des types distincts.

1.5 Décrivez brièvement une implémentation du dictionnaire par une variante des arbres binaires

L'implémentation devra vérifier plusieurs points pour permettre l'implémentation. On vérifie d'abord si l'arbre n'est pas vide. Ensuite, on parcourt l'arbre pour vérifier si la clé existe déjà. Si la clé n'existe pas, on créera un nouveau nœud dans l'arbre de sorte que son prédécesseur le précède au plus près et que la nouvelle sera placée à gauche ou à droite selon un critère défini par la fonction d'abstraction.

1.6 Décrivez brièvement une implémentation du dictionnaire par une variante des arbres R/N

Les arbres RN sont des arbres binaires de recherche (= nœuds internes sont énumérés lors d'un parcours infixe - gauche, nœud, droite - en ordre croissant de clés – le plus petit est le plus à gauche de l'arbre gauche, la plus grande à droite de l'arbre de droite) où chaque nœud est rouge ou noir, la racine est noire, chaque sous-arbre vide est noir, un nœud rouge a 2 enfants noirs et tous les chemins partant de la racine jusqu'à une feuille contiennent le même nombre de nœuds noirs (appelé hauteur noir).

1.6.1 Montrer le caclul des pré- et post-condition de la fonction concrète d'ajout

1.7 Décrivez les 4 types de découpe classique dans "diviser pour régner" avec, pour chaque, un exemple

- Division en un élément et tout le reste - Exemple : Tri par insertion
- Division en deux moitiés égales - Tri par fusion
- Recombination d'un élément avec le reste - Tri par sélection
- Recombination de la moitié avec une autre moitié - Quicksort

1.8 Qu'est ce qu'une fonction d'abstraction pour une implémentation d'un TA ?

Il s'agit d'une fonction qui lie les valeurs concrètes aux valeurs abstraites d'un TA.

1.9 Soient la pré- et post-condition d'une fonction d'un TA. Comment calcule-t-on la pré- et post-condition de l'implémentation de cette fonction ?

Le gros avantage de la spécification d'une fonction abstraite est qu'elle ne présuppose pas du comment son implémentation doit se faire. En effet, les pré- et post- sont des contraintes qui doivent être respectées par la fonction. La pré- et la post- de l'implémentation reprend la pré- et la post- de la fonction d'un TA.

1.10 Qu'appelle-t-on la propriété de sous-solution optimale ?

La propriété de sous-solution optimale est utilisée en programmation dynamique lorsqu'il est possible de diviser un problème en sous-problème en calculant une solution pour chacun de ces sous-problèmes. Ces sous-problèmes sont ensuite recombinées pour former une solution globale optimale au problème de départ.

1.11 Expliquer le principe de la preuve d'optimalité d'un algorithme glouton ?

Pour garantir une solution optimale, il faut:

- Que la propriété de sous-stucture optimale soit vérifiée (c'est à dire qu'il s'agisse d'un choix optimal pour un sous-problème donné);
- Que le choix localement optimal (qui semble le meilleur sur le moment) fasse partie de la solution globalement optimale.

Autrement dit, le choix local doit correspondre à la solution optimale de chaque sous-problème.

1.12 Expliquer le principe de mémoïsation

La mémoïsation est un technique qui consiste à stocker les résultats des calculs afin de ne pas avoir à les refaire lorsque les mêmes *input* sont passés à la fonction.

1.13 Qu'est ce qu'un invariant de représentation ?

Il s'agit d'un ensemble de propositions portant sur les attributs de la classe. Cet invariant permet donc de traduire les contraintes du type abstrait en contraintes sur ces attributs, et donc éliminer des valeurs pour lesquelles la fonction d'abstraction ne donne pas de résultats.

1.13.1 Donnez l'invariant de représentation d'un ensemble fini pour une liste triée

```
// @ invariant list != null;
// @ invariant (\forall int i; i < list.length - 1; list[i] <=
   list[i + 1]);
```

1.13.2 Donnez un sous-programme pour l'intersection de 2 ensembles représentés ainsi que les pré- et post-conditions

```
//@ requires ensemble1 != null
//@ requires ensemble2 != null
//@ ensures \result == (\forall int i; i < (\result).length;
   ensemble1.indexOf(\result.get(i)) != -1
//@ ensures \result == (\forall int i; i < (\result).length;
   ensemble2.indexOf(\result.get(i)) != -1
//@ ensures (\forall int i; i < ensemble1.length;
   (\result).indexOf(ensemble1.get(i)) == -1 ==>
   ensemble2.indexOf(ensemble1.get(i)) == -1
//@ ensures (\forall int i; i < ensemble2.length;
   (\result).indexOf(ensemble2.get(i)) == -1 ==>
   ensemble1.indexOf(ensemble2.get(i)) == -1
public List<E> intersection<E>(List<E> ensemble1, List<E>
   ensemble2) {
    List<E> result = new ArrayList<E>();

    int j = 0;

    for (int i = 0; i < ensemble1.size(); i++) {
        int integer = ensemble1.get(i);
```

```
            while (j <= ensemble2.size() && ensemble2.get(j) <
                integer j++;
            if (ensemble2.get(i).equals(integer))
                result.add(integer);
        }

        return result;
}
```

1.13.3 Quel est l'ordre de grandeur de son temp d'exécution ?

Sachant que la liste est triée, nous n'avons pas à recommencer la lecture de la deuxième liste à partir de 0. Celle-ci ne sera donc lue qu'une seule fois sur l'ensemble des *for*. Nous avons donc un ordre de grandeur de $\mathcal{O}(n)$

1.14 Expliquer le principe de la méthode "générer et tester"

Il s'agit de générer toutes les solutions jusqu'à obtenir celle possédant la propriété recherchée. Pour être sur de bien générer toutes les solutions possibles, la méthode dire de séparation et évaluation se base sur le principe de séparer (de manière récursive) le problème en sous-problèmes de cardinalités inférieures (en imposant des contraintes supplémentaires), tant que la résolution de ceux-ci reste difficile

1.15 Supposons qu'on ajoute dans chaque noeud d'un arbre binaire trié, un champ contenant le nombre d'élément dans le sous-arbre correspondant. Faut-il changer la fonction d'abstraction et/ou l'invariant de données ?

La fonction d'abstraction ne va pas changer. Par contre, l'invariant de données va changer. Il doit indiquer que dans chaque noeud de l'arbre, il existe un champ contenant le nombre d'élément dans le sous-arbre correspondant.

1.15.1 Utilisez cette structure de données pour trouver rapidement le nombre de valeurs dans l'arbre qui se trouve entre deux valeurs données

1.16 Expliquez comment utiliser diviser pour régner pour calculer la puissance x d'un nombre

Si $n \in N$, deux approches sont possibles:

- "1 élément et le reste" (D1) consiste à calculer $x^n = x \times x^{n-1}$, x^{n-1} étant lui même calculé sur le même principe, et ainsi de suite jusqu'au cas de base $x^0 = 1$. La complexité de cette approche étant $\mathcal{O}(n)$
- "Deux moitiés égales" (D2) consiste à calculer $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lceil n/2 \rceil}$, les éléments du produit étant eux-mêmes calculés récursivement jusqu'au cas de base, $x^1 = x$ et $x^0 = 1$. La complexité de cette approche étant $\mathcal{O}(log_2 n)$

1.17 Expliquer la programmation dynamique

La programmation dynamique consiste en la résolution d'un problème avec la méthode "Diviser pour régner", à stocker les valeurs calculées au bout de la partie descendante. Comme ces calculs ont tendance à se répéter, on stocke les ces valeurs dans un tableau indexé et on les récupère au lieu de recalculer ces valeurs.

2 Structure de données : Arbres rouges-noirs

2.1 Quel est l'invariant de données d'un arbre RN ?

1. Un arbre RN est composé de noeud de couleur, ceux-ci sont rouges ou noir;
2. La racine est noir;
3. Les feuilles sont noires;
4. Un noeud rouge ne peut posséder que des enfants de couleur noir;
5. Pour chaque noeud de l'arbre, tous les chemins de celui-ci à une feuille contiennent le même nombre de noeuds noirs.

2.2 démontrez que cet invariant implique que la hauteur d'un arbre RN est logarithmique par rapport au nombre d'éléments

Soit h la hauteur de l'arbre RN. D'après la propriété 4, au moins la moitié des noeuds reliant la racine à une feuille, racine non comprise, doivent être noirs.

Dès lors, la hauteur noire de la racine doit valoir, au moins, $h/2$, et donc :

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ n + 1 &\geq 2^{h/2} \\ log(n+1) &\geq log(2^{h/2}) \\ log(n+1) &\geq h/2 \\ h &\leq 2log(n+1) \end{aligned} \tag{2}$$

2.3 Donnez les idées principales de la procédure qui retire un élément et prouvez l'ordre de grandeur de son temps d'exécution

Suppression du noeud

- z à un enfant gauche (ou droit) null : remplacer par l'enfant droit (ou gauche) de celui-ci
- Aussi non, on fait un translation entre ne plus petit élément "droit" de z et z

Correction de l'invariant de l'arbre

- Cas 1 : le frère de x, w est rouge
  - w devient noir
  - $x.parent$ devient rouge
  - On effectue une rotation gauche sur $x.parent$
- Cas 2 : le frère de x, w est noir et ses deux enfants sont noirs
  - w devient rouge
  - $x = x.parent$
    - Si x est rouge, mettre x en noir
    - Si x est noir, regarder si on est dans le cas 1, 2, 3 ou 4
- Cas 3 : le frère de x, w est noir et l'enfant gauche de w est rouge et l'enfant droit de w est noir

- w devient rouge
- $w.left$ devient noir
- On effectue une rotation droite
- $w = x.parent.right$
- On effectue le cas 4

- Cas 4 : le frère de x, w, est noir et l'enfant de w est rouge
  - w prend la couleur de $x.parent$
  - $x.parent$ devient noir
  - $w.right$ devient noir
  - On effectue un rotation gauche

Après tous les cas, on remet la racine en noir.
Pour la suppression, nous avons les temps d'exécution suivant:

- Pour la suppression en elle-même : $\mathcal{O}(logn)$ à cause de la recherche du plus petit élément de droit
- Pour la correction de l'arbre, nous avons le risque de remonter le problème jusqu'à la racine et donc de , la aussi, parcourir l'ensemble d'une branche. Nous avons donc un temps $\mathcal{O}(logn)$

Le temps pour la suppression est donc logarithmique.

2.4 Quel est le nombre maximal de feuilles d'un arbre RN de hauteur h donnée ?

Le nombre maximal de feuilles est d'un arbre parfait, il est donc : 2^{h+1}. h étant la hauteur d'une branche, la racine ayant pour valeur 0 et les feuilles n'étant pas comprise dedans.

2.5 Quel est le nombre maximal de feuilles d'un arbre binaire de hauteur h donnée ?

Le nombre maximal de feuilles est celui d'un arbre parfait, il y a donc : $n = 2^h$. h étant la hauteur d'une branche, la racine ayant pour valeur 0 et les feuilles étant comprise dedans.

2.6 Quel est le nombre minimal de feuilles d'un arbre RN de hauteur h donnée ?

$2^{\frac{h+3}{2}} - 1 \leq p \leq 2^{h+1}$ donc $2^{\frac{h+3}{2}} - 1$ est le nombre minimale de feuilles pour une hauteur h.

2.7 Quel est le nombre minimal de feuilles d'un arbre binaire de hauteur h donnée ?

Il y a minimum 1 feuilles, pour l'arbre binaire dégénéré, et ceci pour n'importe quelle valeur de h.

2.8 Combien de pointeurs $null$ contient un arbre RN contenant n valeurs ?

$p = n + 1$

2.9 Quel est le nombre maximal de noeuds intérieurs dans un arbre binaire de hauteur h donnée ?

$h \leq n_{internes} \leq 2^h - 1$

2.10 Quel est le nombre minimal de noeuds intérieurs dans un arbre binaire de hauteur h donnée ?

Le nombre minimal de noeud internes est $h - 1$

2.11 Qu'est-ce qu'une rotation gauche d'un arbre binaire ?

Il s'agit d'une opération de transformation d'arbre binaire qui préserve la propriété d'arbre binaire. Pour effectuer une rotation gauche autour d'un noeud z

- Mettre $z.parent = z.right$
- Mettre $t = z.right.left$, $z.right.left = z$, $z.right = t$

2.12 La rotation gauche d'un arbre RN est-elle un arbre RN ?

Non car une rotation modifie l'arbre et le résultat pourrait violer les propriétés d'un arbre RN.

2.13 Quel est l'ordre de grandeur du temps d'une rotation ?

L'ordre de grandeur du temps de rotation est $\mathcal{O}(1)$ car elle modifie un nombre constant de pointeurs.

2.14 Expliquez comment rééquilibrer un arbre RN après une insertion

- Cas 1 : l'oncle y de z (l'élément ajouté) est rouge
  - Mettre $z.parent$ et y à noir
  - Mettre $z.parent.parent$ à rouge
  - Mettre $z.parent.parent$ en tant que nouveau z et recommencer s'il y a toujours une violation
- Cas 2 : l'oncle y de z est noir et z est un fils droit
  - On effectue une rotation gauche (sur z) pour passer dans le cas 3
- Cas 3 : l'oncle y de z est noir et z est un fils gauche
  - On change la couleur de $z.parent^ennoir$ et de $z.parent.parent$ en rouge
  - On effectue une rotation droite (sur $z.parent.parent$)

2.15 Quel est l'intérêt des arbres RN par rapport aux arbres binaires triés ? Prouvez une relation entre leur hauteur et leur nombre d'éléments

L'intérêt des arbres RN par rapport aux arbres binaires triés est la meilleur complexité qu'offre les arbres RN. En effet, pour un arbre binaire trié composé de n noeuds, la hauteur h est au pire $h = n - 1$, ce qui nous donne une complexité de l'ordre de $\mathcal{O}(n)$ pour le parcourir. Alors que pour un arbre RN composé de n noeuds, la hauteur est au pire $h = 2log(n + 1)$ ce qui nous donne une complexité $\mathcal{O}(log(n))$ pour le parcourir.

2.16 Quels sont les pré- et post- de l'insertion dans un arbre RN ?

pre: L'arbre est valide, les invariants sont donc respectés.

post: L'arbre est toujours valide, les invariants sont donc respectés et l'arbre contient en plus l'élément qui a été ajouté.

2.17 Quels sont les pré- et post- de la suppression dans un arbre RN ?

pre: L'arbre est valide, les invariants sont donc respectés.

post: L'arbre est toujours valide, les invariants sont donc respectés et l'arbre ne contient en plus l'élément qui a été supprimé si celui-ci existait.

2.18 Décrivez les arbres RN ? Quel type abstrait représentent-ils ?

Arbre RN est un arbre binaire de recherche $\rightarrow$ adapté à l'implémentation de type "dictionnaire". En particulier, l'indexation d'un grand nombre d'information (ex.: indexation des fichiers dans un O.S.)

2.19 Comment modifier un arbre RN pour représenter le type dictionnaire (map) ?

If suffit d'ajouter un second champ dans le noeud, représentant la valeur x associée à la clé k dans le dictionnaire.

3 Stucture de données : AVL

3.1 Quel est l'invariant de données d'un arbre AVL ?

Pour tout noeud interne, la hauteur du sous-arbre de gauche et de droite ne doit pas différer que d'un au plus.

$$b(T) \in \{-1, 0, 1\}, avec\ b(T) = h(T.right) - h(T.left) \tag{3}$$

$b(T)$ est le facteur d'équilibrage
$h(T)$ calcule la hauteur de l'arbre

3.2 Donner les étapes de l'insertion dans un arbre AVL

L'insertion se passe comme dans un BST, mais l'arbre doit ensuite être rééquilibré.

- Cas gauche-gauche : n_1 est trop déséquilibré à gauche et son enfant gauche est également déséquilibré à gauche : un rotation droite de n_1 suffit.
- Cas gauche-droite : n_1 est trop déséquilibre à gauche et son enfant gauche est déséquilibré à droite : on effectue une rotation gauche sur $n_1.left$ avant d'effectuer une rotation droite sur n_1
- Cas droite-droite : n_1 est trop déséquilibré à droite et son enfant droite est également trop déséquilibré à droite : une rotation gauche de n_1 suffit.
- Cas droite-gauche : n_1 est trop déséquilibré à droite et son enfant droite est déséquilibré à gauche : on effectue un rotation droite sur $n_1.right$ avant d'effectuer une rotation gauche sur n_1

3.3 Donner les étapes de la suppression dans un arbre AVL

La suppression se passe comme dans un BST, mais la suppression d'un noeud peut déséquilibrer l'arbre. Le rééquilibrage se passe comme pour l'insertion.

- Cas gauche-gauche : n_1 est trop déséquilibré à gauche et son enfant gauche est également déséquilibré à gauche : une rotation droite sur n_1 suffit.
- Cas gauche-droite : n_1 est trop déséquilibré à gauche et son enfant gauche et déséquilibré à droite : on effectue une rotation gauche sur $n_1.left$ avant d'effectuer une rotation gauche sur n_1
- Cas droite-droite : n_1 est trop déséquilibré à droite et son enfant droite est également déséquilibré à droite : un rotation gauche de n_1 suffit
- Cas droite-gauche : n_1 est trop déséquilibré à droite et son enfant droite est déséquilibré à gauche : on effectue une rotation droite sur $n_1.right$ avant d'effectuer une rotation gauche sur n_1

3.4 Donnez le nombre de noeuds externes pour un AVL

$$f(h+1) \leq c \leq 2^{h+1} \tag{4}$$

3.5 Donnez le nombre de noeuds internes pour un AVL

$$f(h+2) - 1 \leq q \leq 2^h - 1 \tag{5}$$

3.6 Donnez la hauteur h à partir du nombre de noeuds n

On sait que l'arbre AVL le plus déséquilibré est tel que tout au long de l'arbre, l'un des fils est systématiquement plus haut de 1 par rapport à l'autre. On peut donc caractériser le nombre de noeud en fonction d'un h donné comme suit : $N(h) = 1 + N(h-1) + N(h-2)$.

C'est ce que l'on nomme un arbre de Fibonacci. Dès lors, on a :

$$\begin{aligned} N(h) &= 1 + N(h-1) + N(h-2) \\ N(h) &> 1 + 2N(h-2) \\ N(h) &> \mathcal{O}(2^{(h/2)}) \\ h/2 &< logn \\ h &< 2logn \end{aligned} \tag{6}$$

$$log_2(n+1) \leq h \leq 1.44log_2(n+1) \tag{7}$$