Questions récurrentes

December 28, 2019

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 est un ensemble dynamique d'éléments avec des clés comparables qui supportent différentes opérations comme par exemple $search(S, k)$, $insert(S, k)$, $delete(S, k)$ ou S représente l'ensemble et k un élément du même type que l'ensemble.

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).

1.7 Donnez une implémentation du dictionnaire par un arbre binaire (trié)

1.7.1 Invariant de représentation, invariant de données et fonction d'abstraction

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

1.8 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

1.9 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.

1.10 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.

1.11 Qu'est ce que l'encapsulation d'un TA ?

Le principe de l'encapsulation est de protéger les données en veillant à ce qu'elles ne soient changées que par les sous-programmes de l'interface, ce qui garantit l'invariant des données.

1.12 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.

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

La solution est optimale à condition que :

- Le choix localement optimal fait partie d'une solution globale optimale
- Il s'agit d'un choix optimal pour un sous-problème (sous-structure optimale)

1.14 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.

1.15 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.

1.15.1 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]);
```

1.15.2 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>();

    for (int i = 0; i < ensemble1.length; i++) {
        if (ensemble2.indexOf(ensemble1.get(i)) != -1) {
```

```
                result.push(ensemble1.get(i));
            }
        }

        return result;
}
```

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

Le temps d'exécution va dépendre de la fonction indexOf(). Si celle-ci parcours simplement le tableau à la recherche de l'élément, le temps d'exécution sera $\mathcal{O}(n^2)$

1.16 Comment peut-on relier un invariant de représentation et sa fonction d'abstraction ?

1.17 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

1.18 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.

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

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

1.20 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 Binary Tree

2.1 Décrivez la structure des B-arbres : déclaration et invariant de données