SOEN 331: Introduction to Formal Methods for Software Engineering
Assignment 4 on Algebraic Specifications

Ali Jannatpour

April 6, 2020

Contents

1 General Information

Date posted: Monday April 6^{th}, 2020.

Date due: Tuesday April 21^{st}, 2020, by 23:59.

Weight: 20% of the overall grade.

2 Introduction

Your assignment is to work in teams of 3-4 and produce a formal specification for Binary Search Tree and AVL Tree ADTs using Algebraic Specifications. Each team should designate a leader who will submit the assignment electronically.

3 Ground rules

This is an assessment exercise. You may not seek any assistance while expecting to receive credit. **You must work strictly within your team and seek no assistance for this project (from the instructor, the teaching assistants, fellow classmates and other teams or external help).** Failure to do so will result in penalties or no credit.

4 Your Assignment

Your assignment is given in two parts: 1. The Binary Search Tree (BST) 2. The AVL Tree . The informal descriptions of the two ADTs are given in the following subsections.
In this assignment, you will produce complete algebraic specifications of the two ADTs, each of which are graded independently. All sorts must explicitly be defined or imported. Each of the ADTs come with set of rules that must be implemented by axioms. Number the axioms in your specification to correspond to the rules. You may need to add additional axioms to address the "hidden" rules that are addressed in the informal description. Once you finalize the formal specification, answer the additional questions underneath each ADT.

4.1 The Binary Search Tree (BST)

The *Binary Search Tree* (BST), is a particular type of rooted binary tree whose internal nodes each store a key and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the *binary search property*, as follows:

> The key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.

In this section you produce a complete algebraic specification of a BST that contains *Key-Value* elements, where Key is a unique *Number*, and Value is a generic type. An informal description of the operations of the BST tree is given below. The Binary Search Tree ADT will be referred to as *BSTree*.

4.1.1 BST Functionalities

- *create* creates an empty binary search tree.
- *add(BSTree, Number, Element)* inserts an element into the right position within the search tree and returns an updated tree. If the key already exists, the add operation fails.
- *update(BSTree, Number, Element)* updates the value for key and returns an updated tree. If the key is not found, the update operation fails.
- *delete(BSTree, Number)* deletes an element from the binary search tree and returns an updated tree. If the key is not found, the delete operation fails.
- *containsKey(BSTree, Number)* checks whether the given key exists in the tree; it returns true if the key exits, otherwise it returns false.
- *contains(BSTree, Element)* checks whether the given element may be found in the values for each node of the tree; it returns true if the key exits, otherwise it returns false.

4.1.2 Tree Functionalities

In additional to above, the BSTree also supports standard binary-tree functionalities, as listed in the following.

- *left(BSTree, Number)* takes a binary search tree and returns the left subtree of the element, whose key is specified by the number; it fails if the key is not found. The returned subtree is a BStree. If the left subtree does not exists, the method returns an empty tree.
- *right(BSTree, Number)* takes a binary search tree and returns the right subtree of the element, whose key is specified by the number; it fails if the key is not found. The returned subtree is a BStree. If the right subtree does not exists, the method returns an empty tree.
- *parent(BSTree, Number)* takes a binary search tree and returns the parent (key) of the element, whose key is specified by the number; it fails if the key is not found.
- *isLeaf(BSTree, Number)* takes a binary search tree and returns true if the node, whose key is specified by the number is a leaf, otherwise it returns false; it fails if the key is not found.
- *rootKey(BSTree)* takes a binary search tree and returns the key that is stored in the root of the tree; it fails if the tree is empty.
- *isEmpty(BSTree)* takes a binary search tree and returns true if the tree is empty, otherwise it returns false.
- *depth(BSTree)* returns the depth of the tree.
- *nNodes(BSTree)* returns the total number of nodes of the tree.
- *nLeaves(BSTree)* returns the total number of leaf nodes of the tree.

4.1.3 Rules

Given the above informal description of the BSTree, provide the formal algebraic specification for the sort. Implement additional axioms to address the following rules. Do NOT repeat axioms.

1. The primitive constructor creates an empty tree. Specify this in two different ways.
2. There is no root defined for an empty tree.
3. The depth of the newly created tree is zero.
4. The left method cannot be applied on a newly created tree.
5. Adding $\langle 1, el_1 \rangle$ to an empty tree results in a tree whose depth is 1.
6. The total number of nodes of the resulting tree in the previous item is 1.
7. Adding one more time of $\langle 1, el_1 \rangle$ to the resulting tree in the previous item fails.
8. The maximum depth of a BST tree equals the number of nodes of a tree.
9. Given a tree has at least one node, the depth of a BST tree is always greater that the logarithm of the number of nodes of a tree in base 2.

For the following rules, $\langle 1, el_1 \rangle$, $\langle 2, el_2 \rangle$, and $\langle 3, el_3 \rangle$ are added to a newly created tree, in the order they have been specified.

10. The depth of the resulting tree is 3.
11. The total number of nodes of the resulting tree is 3.
12. The key of the root of the tree is 1;
13. The right subtree of the root node has two nodes.
14. The left subtree of the root node is empty.

Note: Do not forget to address the "hidden" rules in section 4.1.2.

4.1.4 Additional Questions

Given your formal specification, answer the following questions:

1. How do you define a *deleteAll* method that deletes all the keys and elements of a given BSTree? Formally specify it.

2. How do you define a *getAllKeys* method that find all keys within the BST that have a specific associated value? Formally specify it.

3. Define axioms that verify the clause "*isLeaf(BSTree, Number)* takes a binary search tree and returns true if the node, whose key is specified by the number is a leaf, otherwise it returns false".

4. Define an axiom that verifies the clause "*rootKey(BSTree)* takes a binary search tree and returns the key that is stored in the root of the tree".

5. Define axiom(s) that verify the functionality of the *left(BSTree, Number)* method for the cases: "if the left subtree does not exists, the method returns an empty tree".

6. Define an axiom that verifies a correct key is returned by *parent(BSTree, Number)*.

4.2 The AVL Tree

The *AVL Tree*, named after inventors **A**delson-**V**elsky and **L**andis, is a binary search tree that is self-balancing. An informal description of the ADT, namely *AVLTree* is given below.

4.2.1 Description

The AVLTree subclassifies BSTree and implements / modifies the following operations:

- *create* creates an empty AVL tree.

- *add(AVLTree, Number, Element)* inserts an element into the tree and makes sure the tree is balanced. If the key already exists, the add operation fails.

- *delete(AVLTree, Number)* deletes an element and makes sure the tree is balanced. If the key is not found, the delete operation fails.

All other inherited operations remain unchanged, except for the following operation which modifies the signature of the method:

- *update*: receives a AVLTree along with the other parameters and returns an AVLTree.

Additionally, the following operations are implemented by the AVLTree:

- *balanceFactor(AVLTree)* returns the balance factor of an AVLTree, which is height of a right sub-tree - the height of the left sub-tree.
- *join(AVLTree, AVLTree)* receives two AVL trees and returns a new tree that is created by joining the two trees.

4.2.2 Rules

The following rules are applied on AVLTree:

1. Given an AVLTree with a depth of d, the number of nodes cannot exceed $2^d - 1$.
2. The balance factor of a newly created AVLTree is 0
3. The balance factor of an AVLTree with one node is 0
4. The balance factor of an AVLTree with two nodes is either 1 or -1
5. The balance factor of an AVLTree is always of of these values: $\{-1, 0, 1\}$

Inserting $\langle 1, el_1 \rangle$, $\langle 2, el_2 \rangle$, and $\langle 3, el_3 \rangle$, consecutively into an empty AVLTree, results in the following:

6. The depth of the resulting tree is 2.
7. The total number of nodes in the resulting tree is 3.
8. The balance factor of the resulting tree is 0.

9. The key of the root of the tree will be 2.

10. The right and left subtrees of the root node have both one node in total.

Given the above informal description of the AVLTree, provide the formal algebraic specification for the sort.
Use preconditions to make sure the key operations (add, update, and delete) work without errors.

4.2.3 Additional Questions

Given your formal specification, answer the following questions:

1. How do you define a method that verifies if a given tree is AVLTree?

2. Identify classes (categories) of all operations defined in your specification for AVLTree.

5 What to submit

You must write your specification using the LaTeX text formatting package. Similar to previous assignments, you need to install a) the LaTeX package for your operating system, and b) some LaTeX editor. One recommended combination is `MiKTeX` with `Texmaker`. You will find necessary resources on the course website.
Use the template provided to prepare the formal specification and produce a **pdf** file named after the id of the person to submit, e.g. `123456.pdf`.

Section Submission

Section S Please submit on Moodle. Your instructor will provide you with more details.

Section U Please submit your pdf file at the Electronic Assignment Submission portal

`https://fis.encs.concordia.ca/eas`

under Theory Assignment 4.

References

1. https://en.wikipedia.org/wiki/Binary_search_tree

2. https://en.wikipedia.org/wiki/AVL_tree