Tutorial 09 – Tree Part 2

Introduction

This tutorial focuses on Binary Tree implementation and its applications.
Example 01 demonstrates how to implement a binary tree based on recursive definition.
Example 02 shows the implementation of a binary search tree.
In the exercise section, students are asked to:

- implement some additional operators on binary tree
- solve the finding duplicated problem using a binary search tree
- implement an expression tree, evaluate an expression tree and build an expression tree from an infix expression.

Examples

1. Example 01 – Binary Tree implementation

The basic concept of binary tree implementation based on its recursive definition is introduced in lecture 09. In this example, we consider a binary tree which has node labels of string data type.
The following operators are implemented:

- buildTree(String *rootValue*, BinaryTree *left*, BinaryTree *right*) : void – create a binary tree with the *rootValue* as the value of the root node, and *left* and *right* as the left sub-tree and the right sub-tree.
- buildTreeByValue(String *rootValue*, String *leftValue*, String *rightValue*) : void – create a binary tree with the *rootValue* as the value of the root node, *leftValue* as the value of the left sub-tree, *rightValue* as the value of the right sub-tree.
- isEmpty() : boolean – returns true if the current binary tree is empty (the root, the left sub-tree and the right sub-tree are null), otherwise the operation will return false.
- makeEmpty() :void – makes the binary tree become empty.
- getTreeValue() : String – returns the tree value (the label of the root node).
- setTreeValue(String *label*) : void – set tree value to the new *label* argument.
- isLeaf() : boolean – returns true if the current tree is a leaf tree (contains no sub-trees), otherwise the operation will return false.
- getLeftSubTree() : BinaryTree – returns the left sub-tree of the current binary tree.
- getRightSubTree(): BinaryTree – returns the right sub-tree of the current binary tree.
- setLeftSubTree(BinaryTree *left*) : void – sets the left sub-tree to *left*.
- setRightSubTree(BinaryTree *right*) : void – sets the right sub-tree to *right*.
- preOrderTraversal(BinaryTree *t*) : void – traverses the tree in the pre-order.
- inOrderTraversal(BinaryTree *t*): void – traverses the tree in the in-order.
- postOrderTraversal(BinaryTree *t*): void – traverses the tree in the post-order.
- getDepth(BinaryTree *t*) : int – returns the depth of the current binary tree.
- countLeaves(BinaryTree *t*) : int – returns the total leaves of the current binary tree.
- iPathLength(BinaryTree *t*) : int – returns the internal path length (IPL) of the current binary tree.
- countNodes(BinaryTree *t*) : int – returns the total nodes of the current binary tree.
- clone(BinaryTree *t*) : BinaryTree – returns a copy of the input binary tree *t*.
- isEqual(BinaryTree *t1*, BinaryTree *t2*) : boolean – returns true if *t1* and *t2* are equal, otherwise the operation will return false.

Please refer to class ***BTNode, BinaryTree*** and ***BinaryTreeApp*** in the tutorial source code project.

2. Example 02 – Binary Search Tree

This example demonstrates how to implement a binary search tree. The label (or value) of each node in the tree is an integer. The following operators are implemented:

- addRoot(int *value*) : void – adds the *root* (with *value*) to an empty binary search tree.
- insert(int *key*, BinarySearchTree *t*) : boolean – inserts a new sub-tree with the *key* value into the binary search tree *t*. If the current binary search tree is empty, then *key* will be assigned as the root value of the tree. If the *key* value already exists in the tree, the the operation will return false, otherwise the operation will return true.
- getMax() : int – returns the maximum value of all nodes in the tree.
- getMin() : int – returns the minimum value of all nodes in the tree.
- search(int key) : boolean – searchs the tree for the *key* value. The operation will return true if the key exists in the current tree, otherwise the operation will return false.
- preOrderTraversal(BinarySearchTree *t*) : void – traverses the tree in the pre-order.

Please refer to class ***BSTNode, BinarySearchTree*** and ***BinarySearchTreeApp*** in the tutorial source code project.

Exercises

1. Exercise 1

Please implement the following operations on the binary tree:

- inOrderTraversal(BinaryTree *t*): void – traverses the tree in the in-order.
- postOrderTraversal(BinaryTree *t*): void – traverses the tree in the post-order.
- clone(BinaryTree *t*) : BinaryTree – returns a copy of the input binary tree *t*.
- isEqual(BinaryTree *t1*, BinaryTree *t2*) : boolean – returns true if *t1* and *t2* are equal, otherwise the operation will return false.

2. Exercise 2

Please write a program to solve the finding duplicate problem presented in the lecture 10. Your program should:

- Ask user to input a list of *N* integers using keyboards. These integers will be stored in an array *A*.
- Build a binary search tree *T* from the array *A* using the class ***BSTNode*** and ***BinarySearchTree*** above.
- Show all duplicate items in *A* when building *T*.

3. Exercise 3

Please write a program to work with the expression trees presented in the lecture 10. Your program should:

- Ask user to input an expression *E* in infix notation (with the parentheses).
- Converts *E* to the post-fix form.
- Build an expression tree *T* corresponding to *E*. An expression tree is a binary tree that each node of the tree has a String label. You must create two class ***ETNode*** and ***ExpressionTree*** to implement the expression tree data structure. An expression tree stack must also be implemented to support the building task.
- Evaluate *T* and display the result.