UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS212 - Data structures and algorithms

Practical 3 Specifications: Binary Search Tree (Simple Traversals) and Recursion

Release Date: 13-03-2023 at 06:00

Due Date: 17-03-2023 at 23:59

Total Marks: 361

Contents

1 General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may **not** import any provided Java library. Importing any library or data structure will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire specification before you start coding.
- You will be afforded three upload opportunities.
- Submissions that result in a compilation failure or a run time error will also receive a mark of zero.

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The primary goal of this assignment is to implement a set of Binary Search Tree (BST) traversal algorithms and gain experience with recursion.

4 Introduction

Complete the task below. Certain classes have been provided for you alongside this specification in the Student Files folder. A very basic main has been provided. **Please note this main is not extensive and you will need to expand on it**. Remember to test boundary cases. Submission instructions are given at the end of this document.

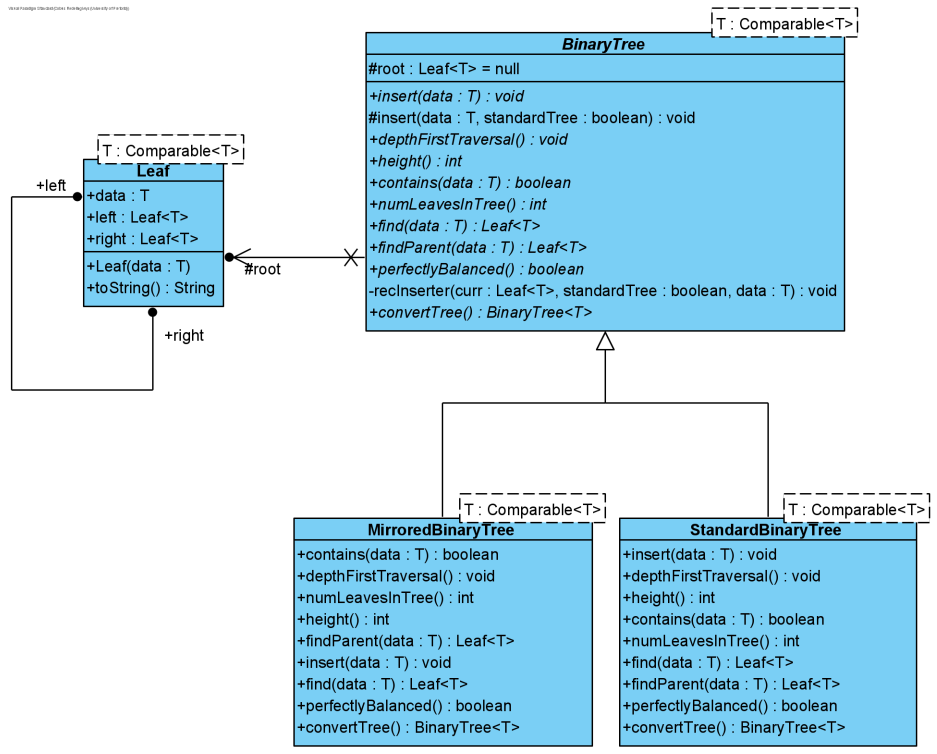
For this practical, you will implement two BSTs. One will be a standard BST where the left child has a smaller value than the right child. The other will be a mirrored BST where the right child has a smaller value than the left child.
You may only use recursion for this practical. The use of any iterative solution is prohibited. The use of the following keywords is also prohibited:

- for
- do
- while

Violating this restriction will result in a mark of 0.

5 Task:

Your task for this practical will be to implement the following class diagram as described in later sections.



T : Comparable<T>
BinaryTree
#root : Leaf<T> = null
+insert(data : T) : void
#insert(data : T, standardTree : boolean) : void
+depthFirstTraversal() : void
+height() : int
+contains(data : T) : boolean
+numLeavesInTree() : int
+find(data : T) : Leaf<T>
+findParent(data : T) : Leaf<T>
+perfectlyBalanced() : boolean
-recInserter(curr : Leaf<T>, standardTree : boolean, data : T) : void
+convertTree() : BinaryTree<T>
T : Comparable<T>
Leaf
+data : T
+left : Leaf<T>
+right : Leaf<T>
+Leaf(data : T)
+toString() : String
+left
+right
#root
T : Comparable<T>
MirroredBinaryTree
+contains(data : T) : boolean
+depthFirstTraversal() : void
+numLeavesInTree() : int
+height() : int
+findParent(data : T) : Leaf<T>
+insert(data : T) : void
+find(data : T) : Leaf<T>
+perfectlyBalanced() : boolean
+convertTree() : BinaryTree<T>
T : Comparable<T>
StandardBinaryTree
+insert(data : T) : void
+depthFirstTraversal() : void
+height() : int
+contains(data : T) : boolean
+numLeavesInTree() : int
+find(data : T) : Leaf<T>
+findParent(data : T) : Leaf<T>
+perfectlyBalanced() : boolean
+convertTree() : BinaryTree<T>

Figure 1: Class diagram

5.1 Leaf

This will be the class that represents the leaves in the Binary Search Trees.

- Members:
  - data: T
    - This is the variable that will contain the data for the leaf.
  - left: Leaf<T>
    - This is the left child for the current leaf.
    - Remember this member variable can be null.

  - right: Leaf<T>
    - This is the right child for the current leaf.
    - Remember this member variable can be null.

- Functions:
  - Leaf(data: T)
    - This is the constructor for the Leaf class.
    - Initialize the appropriate member variable with the passed-in parameter.
    - Initialize all other member variables to null.
  - String: toString()
    - This is a provided function.
    - Do not alter this function.
    - Altering this function will negatively impact your marks.

5.2 BinaryTree

This is the base class for the binary tree hierarchy.

- Members:
  - root: Leaf<T>
    - This is the root of the tree.
    - The root should have a value of null until it is initialized.
- Functions:
  - insert(data: T): void
    - This is an abstract function that will be used to insert values into the tree.
  - insert(data: T, standardTree: boolean): void
    - This is a provided function.

    * Do not alter this function.
    * Altering this function will negatively impact your marks.
- depthFirstTraversal: void
    * This is an abstract function.
    * This function will traverse the tree in a depth-first inorder manner.
    * The function should print out each leaf on a new line in a depth-first inorder manner.
    * *Hint: Use the leaf's toString function*
- height(): int
    * This is an abstract function.
    * This function will return the height of the tree.
    * Note that root has a height of 0.
    * Example: The tree in Figure 2 has a height of 2.
- contains(data: T): boolean
    * This is an abstract function.
    * This function will determine if a leaf in the tree has a data member variable that is equal to the passed-in parameter or not.
    * Use Leaf's toString to get the string representation of the leaf and print out each leaf it visits on a new line as it traverses down the tree to determine the answer.
    * Note the path should be the shortest possible path.
    * If the data is not in the tree, the function should return false, otherwise it should return true.
- numLeavesInTree(): int
    * This is an abstract function.
    * This function will determine how many leaves are currently in the tree.

    * If the tree is empty the function should return 0.
    * Example: The tree in Figure 2, has 7 leaves.
- find(data: T): Leaf<T>
    * This is an abstract function.
    * This function should return the leaf that contains the same data member variable as the passed-in parameter.
    * Use Leaf's toString to get the string representation of the leaf and print out each leaf it visits on a new line as it traverses down the tree to determine the answer.
    * Note the path should be the shortest possible path.
    * If there is no leaf that contains the data, the function should return null but the path should still be printed.
- findParent(data: T): Leaf<T>
    * This is an abstract function.
    * This function should return the parent of the leaf that contains the same data member variable as the passed-in parameter.
    * Use Leaf's toString to get the string representation of the leaf and print out each leaf it visits on a new line as it traverses down the tree to determine the answer.
    * Note the path should be the shortest possible path.
    * If the data belongs to the root leaf, the function should return null. In this special case, the printed path should only contain the root node.
    * In the standard case the path should not contain the leaf with the data member. In other words, the last leaf that should be printed in the parent leaf.
    * If no leaf in the tree contains the data, the function should also return null but the path should still be printed.

- perfectlyBalanced(): boolean
  - This is an abstract function.
  - For this practical a Binary Tree is considered perfectly balanced if, for all of the leaves, the number of leaves in the left sub-tree is the same as the number of leaves in the right sub-tree.
  - This function should determine if the tree is perfectly balanced or not, using the definition given above.
  - If the tree is perfectly balanced, the function should return true, othwerwise it should return false.
  - Figures 2 and 3 are examples of perfectly balanced trees.
- recInserter(curr: Leaf<T>. standardTree: boolean, data: T): void
  - This is a provided function.
  - Do not alter this function.
  - Altering this function will negatively impact your marks.
- convertTree(): BinaryTree<T>
  - This is an abstract function.
  - This function will convert a StandardBinaryTree into a MirroredBinaryTree and visa versa.
  - This function will return the newly created tree.
  - The result that should be returned if convertTree is called on Figure 2 is Figure 3
  - The result that should be returned if convertTree is called on Figure 3 is Figure 2
  - If the tree is empty, an empty tree should be returned.

5.3 StandardBinaryTree

This class inherits from the BinaryTree base class.

- Functions:
  - insert(data: T): void
    - This is a provided function.
    - Do not alter this function.
    - Altering this function will negatively impact your marks.
  - depthFirstTraversal(): void
    - This function should traverse the tree as described in the above section.
    - The function should first go to the left child and then the right child.
    - Example: Given the tree in Figure 2. The order of leaves printed out should be: $[1; 3; 4; 5; 6; 7; 8]$
    - Remember to use the toString function to print out each leaf on a new line.
  - height(): int
    - Implement the function as described in the above section.
  - contains(data: T): boolean
    - Implement the function as described in the above section.
    - Example: Given the tree in 2. The order of leaves printed out should be as if 4 is to be found: $[5, 3, 4]$
    - Remember to use the toString function to print out each leaf on a new line.
  - numLeavesInTree(): void
    - Implement the function as described in the above section.
  - find(data: T): Leaf<T>
    - Implement the function as described in the above section.

    - Example: Given the tree in Figure 2. The order of leaves printed out should be as if 4 is to be found: $[5, 3, 4]$
    - Remember to use the toString function to print out each leaf on a new line.
  - findParent(data: T): Leaf<T>
    - Implement the function as described in the above section.
    - Example: Given the tree in Figure 2. The order of leaves printed out should be as if 4's parent is to be found: $[5, 3]$
    - Remember to use the toString function to print out each leaf on a new line.
  - perfectlyBalanced(): boolean
    - Implement the function as described in the above section.
  - convertTree(): BinaryTree<T>
    - Implement the function as described in the above section.

5.4 MirroredBinaryTree

This class publicly inherits from the BinaryTree base class

- Functions:
  - insert(data: T): void
    - This is a provided function.
    - Do not alter this function.
    - Altering this function will negatively impact your marks.
  - depthFirstTraversal(): void
    - This function should traverse the tree as described in the above section.
    - The function should first go to the **right child** and then the **left** child.
    - Example: Given the tree in Figure 3. The order of leaves printed out should be: $[1, 3, 4, 5, 6, 7, 8]$

    * Remember to use the toString function to print out each leaf on a new line.

- height(): int
    * Implement the function as described in the above section.
- contains(data: T): boolean
    * Implement the function as described in the above section.
    * Example: Given the tree in Figure 3. The order of leaves printed out should be as if 4 is to be found: $[5, 3, 4]$
    * Remember to use the toString function to print out each leaf on a new line.
- numLeavesInTree(): void
    * Implement the function as described in the above section.
- find(data: T): Leaf<T>
    * Implement the function as described in the above section.
    * Example: Given the tree in Figure 3. The order of leaves printed out should be as if 4 is to be found: $[5, 3, 4]$
    * Remember to use the toString function to print out each leaf on a new line.
- findParent(data: T): Leaf<T>
    * Implement the function as described in the above section.
    * Example: Given the tree in Figure 3. The order of leaves printed out should be as if 4's parent is to be found: $[5, 3]$
    * Remember to use the toString function to print out each leaf on a new line.
- perfectlyBalanced(): boolean
    * Implement the function as described in the above section.
- convertTree(): BinaryTree<T>
    * Implement the function as described in the above section.

6 Hint

When the spec says to print out the leaf on a new line, the following is implied:

```
    System.out.println(leaf.toString());
```

7 Helper functions and variables

You are allowed to add helper functions to any of the classes. You are strongly encouraged to add your own helper recursive functions.

8 Restrictions for all classes

As a reminder, you may only use recursion for the implementation of all the functions in the practical. The following keywords are thus illegal to use in this practical:

- for
- while
- do

Usage of these commands in any of the classes will yield a mark of 0 for the entire practical.

9 Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. The following java files should at least be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- BinaryTree.java
- Leaf.java
- MirroredBinaryTree.java
- StandardBinaryTree.java

You may add any other custom classes that you created. Your code should be able to be compiled with the following command:

```
make *.java
```

and run with the following command:

```
java App
```

You have 5 submissions and your best mark will be your final mark. Upload your archive to the Practical 3 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

10 Example Figures

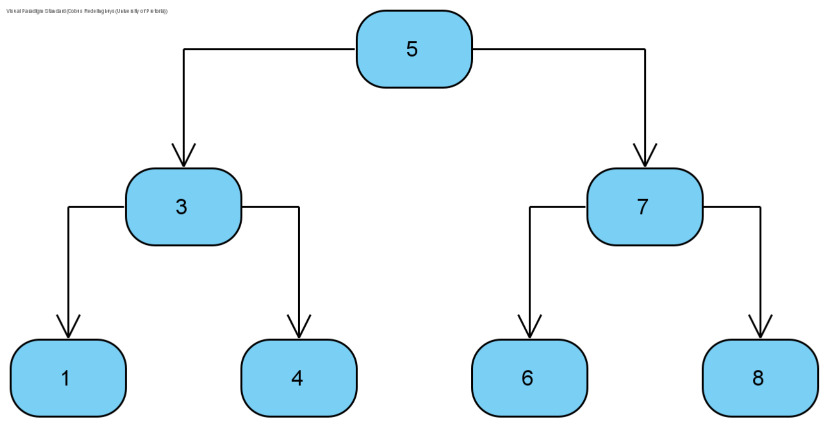

Visual Paradigm Standard (Cobus Redelinghuys (University of Pretoria))
5
3
7
1
4
6
8

Figure 2: Example 1: Standard BST

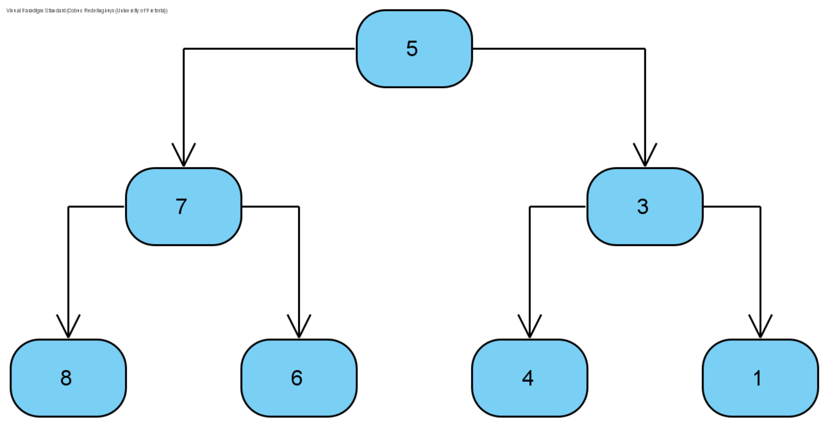

Visual Paradigm Standard (Cobus Redelinghuys (University of Pretoria))
5
7
3
8
6
4
1

Figure 3: Example 2: Mirrored Binary Tree