



BTRecurTutor: a tutorial for practicing recursion in binary trees

Sally Hamouda , Stephen H. Edwards , Hicham G. Elmongui , Jeremy V. Ernst & Clifford A. Shaffer


To cite this article: Sally Hamouda , Stephen H. Edwards , Hicham G. Elmongui , Jeremy V. Ernst & Clifford A. Shaffer (2020) BTRecurTutor: a tutorial for practicing recursion in binary trees, Computer Science Education, 30:2, 216-248, DOI: [10.1080/08993408.2020.1714533](https://doi.org/10.1080/08993408.2020.1714533)

To link to this article: <https://doi.org/10.1080/08993408.2020.1714533>



Published online: 20 Jan 2020.



[Submit your article to this journal](#) 



Article views: 67



[View related articles](#) 



[View Crossmark data](#) 



BTRecurTutor: a tutorial for practicing recursion in binary trees

Sally Hamouda^a, Stephen H. Edwards^b, Hicham G. Elmongui ^c, Jeremy V. Ernst^d
and Clifford A. Shaffer^b

^aDepartment of Math/CS, Rhode Island College, Providence, RI, USA; ^bDepartment of Computer Science, Virginia Tech, Blacksburg, VA, USA; ^cDepartment of Computer and Systems Engineering, Alexandria University, Alexandria, Egypt; ^dDepartment of Social Sciences & Economics, Embry-Riddle University, Daytona Beach, FL, USA

ABSTRACT

Background and Context: Recursion in binary trees has proven to be a hard topic. There was not much research on enhancing student understanding of this topic.

Objective: We present a tutorial to enhance learning through practice of recursive operations in binary trees, as it is typically taught post-CS2.

Method: We identified the misconceptions students have in recursive operations on binary trees. We designed a code writing exam question to measure those misconceptions. We built a tutorial that trains students on avoiding those misconceptions through the use of a semantic code analyzer that detects misconceptions and provides appropriate feedback.

Findings: Our results show an improvement in student performance when using the tutorial along with the practice exercises, and even more improvement when the same exercises are used with a semantic code analyzer.

Implications: The best way to use our tutorial to enhance student performance on advanced recursion is to allow students solving the tutorial exercises with the the semantic feedback.

ARTICLE HISTORY

Received 29 April 2019
Accepted 8 January 2020

KEYWORDS

Recursion; binary Trees;
auto-graded Exercises;
misconceptions

1. Introduction

This paper presents our efforts to enhance the learning of recursion as it is typically taught post-CS2. This is often cast in the form of recursive operations on binary trees. Recursion in binary trees has proven to be a hard topic that students struggle with (Grissom, Murphy, McCauley, & Fitzgerald, 2016; Murphy, Fitzgerald, Grissom, & McCauley, 2015). While there has been a lot of attention given to teaching introductory recursion, there was previously not much research on enhancing student understanding of recursion at an intermediate level. Existing work has identified student misconceptions in the context of

binary trees, but this was not related to recursion (Karpierz & Wolfman, 2014). What work we do know of related to recursion in binary trees addresses generic problems in writing recursive functions, not specific misconceptions regarding recursion in binary trees.

We have conducted student interviews and analyzed many student answers to code writing questions on recursion in binary trees to come up with a list of the common misconceptions held by students for this topic. We present our findings from the interviews and the analysis of student answers as a list of misconceptions and difficulties, inspired by Ragonis and Ben-Ari's work on object oriented programming (Ragonis & Ben-Ari, 2005). A misconception is a mistaken idea or view resulting from a misunderstanding of something. By "difficulty" we mean the empirically observed inability to do something. It is possible that a student exhibits a difficulty due to an underlying misconception (possibly one already listed here or one so far unidentified). A difficulty might also result because the student lacks some skill or knowledge.

We have built a tutorial that addresses those misconceptions and trains students on avoiding them. The tutorial features code writing questions. It trains students in part through feedback generated by a semantic code analyzer that detects misconceptions in the students' answers to the practice exercises. To help assess our work, we have designed a set of questions that could be used as a pre-test or post-test to measure those misconceptions. We have designed a rubric for each question to match possible answers to misconceptions.

Our goal is to answer the following research questions:

- (1) What is the effect of forcing students to practice more programming exercises built to address misconceptions regarding recursion on binary trees on student performance?
- (2) What is the effect of warning students about their misconceptions on student performance for this topic?

2. Related work

2.1. Recursion

In our previous work Hamouda, Edwards, Elmongui, Ernst, and Shaffer (2017) we documented prior literature on the teaching of recursion. For more details, see the literature review section there. Some major works include Chaffin, Doran, Hicks, and Barnes (2009); Ginat and Shifroni (1999); Scholtz and Sanders (2010); Tessler, Beth, and Lin (2013); Wilcocks and Sanders (1994) and Tessler et al. (2013). We note that none of these studies adopted the pedagogical model of providing an interactive tutorial combined with practice on programming exercises that address recursion in binary trees misconceptions with automated assessment and feedback to the students. The only work that we are aware of

related to recursion in binary trees was by Murphy et al. (2015). They conducted a goal-plan analysis to find the plans used by students when writing a recursive method to count the number of nodes that have exactly one child in a Binary Search Tree. Analysis of student answers showed that the most common errors for the tree traversal goal were missing and malformed base cases. Even some students who did not have difficulty with base cases would misplace calculations and miss recursive calls. Murphy et al. found their data to be useful for designing questions. This work does not identify or correct specific misconceptions in binary trees. Instead, it is related to general difficulties in writing a recursive method, like writing a correct base case or recursive call. Previous work on finding and addressing misconceptions related to recursion in binary trees was limited. However, there have been efforts to find and address misconceptions related to other CS topics. Karpierz and Wolfman (2014) report an initial effort to determine misconceptions and design a CI for Binary Search Trees and Hash Tables. They focused on iterative methods rather than recursion. The authors found student misconceptions by showing exam responses to nine instructors, with the goal to understand how an expert recognises something important that novices do not. The authors also reviewed more than 200 exam problems along with project code to determine the most difficult problems. They interviewed 25 students who each solved two questions while thinking aloud. The authors found three main topics where students hold misconceptions: the possibility of duplicates in BSTs, conflation of Heaps and BSTs, and hash table resizing. The authors designed three multiple choice questions to address those misconceptions.

2.2. Misconceptions and concept inventories

There is limited previous work on finding and addressing misconceptions related to recursion in binary trees was limited. However, there have been efforts to find and address misconceptions related to other CS topics.

Karpierz and Wolfman (2014) report an initial effort to determine misconceptions and design a Concept Inventory for Binary Search Trees and Hash Tables. A Concept Inventory (CI) is a test that can classify an examinee as either someone who thinks in accordance with accepted conceptions on a body of knowledge or in accordance with common misconceptions (Adams & Wieman, 2011; Rowe & Smail, 2007). Karpierz and Wolfman focused on iterative methods rather than recursion. The authors found student misconceptions by showing exam responses to nine instructors, with the goal to understand how an expert recognises something important that novices do not. The authors also reviewed more than 200 exam problems along with project code to determine the most difficult problems. They interviewed 25 students who each solved two questions while thinking aloud. The authors found three main topics where students hold misconceptions: the possibility of duplicates in BSTs, conflation of Heaps and

BSTs, and hash table resizing. The authors designed three multiple choice questions to address those misconceptions.

Kaczmarczyk, Petrick, East, and Herman (2010) document student misconceptions in a CS1-level programming course. Using a Delphi process (Dalkey & Helmer, 1963), the authors' pool of experts determined 30 concepts that they think are the most difficult in CS1 programming. From these the authors selected ten concepts as their initial focus of interest. They are: memory model, references and pointers, primitive and reference type variables, control flow, iteration and loops, types, conditionals, assignment statements, arrays, and operator precedence. The authors designed a test of 18 questions covering the concepts of interest. In order to make sure that the results are not problem dependent, each concept was covered in questions with at least two different variations. The authors conducted student interviews to help them understand student misconceptions regarding the targeted concepts. Eleven undergraduate students participated in the interviews. These students were either currently or recently enrolled in the CS1 course. Each interview lasted about an hour and was audio and video recorded. In the interviews, each student was asked to solve questions for all ten concepts. The purpose of the interview was to reveal the misconceptions of the students and validate the expert pool's conclusions about the difficult concepts. The authors analyzed the student interviews and described in detail the misconceptions found in memory model representation and default value assignment of primitive values.

Danielsiek, Paul, and Vahrenhold (2012) described their first steps towards building a concept inventory (CI) for Algorithms and Data Structures. Their results were based on expert interviews and the analysis of 400 exams to identify the core concepts that are considered to be associated with misconceptions. They reported a pilot study to verify misconceptions previously reported in the literature, and to identify additional misconceptions. They have then wrote an initial instrument to detect misconceptions related to algorithms and data structures (Paul & Vahrenhold, 2013). They presented the results from a second study that aimed at assessing first-year student misconceptions. Their second study confirmed findings from the previous small-scale studies, but additionally broadened the scope of the topics.

Ragonis and Ben-Ari (2005) presented an initial effort to identify misconceptions and difficulties in object-oriented programming (OOP). The authors gathered data during two academic years from students studying OOP in tenth grade CS. The data gathered included home works, lab exercises, tests, and projects. They used these data to identify a comprehensive categorized list of misconceptions and difficulties in OOP understanding. One novel aspect of this work is the reporting of difficulties in addition to misconceptions.

Taylor et al. (2014) presented a survey paper on Computer Science concept inventories. It includes a recommendation to build CIs for topics that should evaluate student's ability to engage in processes such as code analysis, program

design, program modification, and testing, as these aspects of learning are difficult to assess. Similarly, Zingaro, Petersen, and Craig (2012) notes that it is hard to evaluate traditional code writing exercises.

2.3. Automated assessment of programming exercises

Previous research has shown that automated assessment of programming exercises is beneficial for both students and instructors (Laakso, Salakoski, & Korhonen, 2005; Saikkonen, Malmi, & Korhonen, 2001). Saikkonen et al. (2001) implemented Scheme-robot for assessing programming exercises written in the functional programming language Scheme. Scheme-robot assesses individual procedures instead of complete programs and provides feedback to the students. At the time of their publication, the system had been used in an introductory programming course with some 350 students for two years. Students practiced five exercises on average per week without putting work on the instructor to correct their answers. Also, students got immediate feedback on their answers, which helped them to learn from their mistakes. The authors conducted a survey showing that 80% of the students thought that automatic assessment, in general, is a good or an excellent idea.

Laakso et al. (2005) studied the feasibility of automatically assessed exercises. The authors recommend using both in-class and automatically assessed exercises instead of using only one of these.

There are multiple approaches used for automated assessment of programming exercises in general. These include output-based (also known as dynamic), static, and trace-based assessment.

2.3.1. Output-based program assessment

Output-based program assessment runs a program against test cases, then compares the output of the student's code against output from a model answer. An example of output-based testing is unit testing. Unit testing is a level of software testing where individual components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program or function. In object-oriented programming, the smallest unit is a method which belongs to a class.

There has been much work on output-based program assessment. For example, Blumenstein, Green, Nguyen, & Muthukkumarasamy (2004); Helmick (2007); Karavirta and Ihantola (2010); Llana, Martin-Martin, Pareja-Flores, and Velázquez-Iturbide (2014); Longo, Sterbini, and Temperini (2009); Saikkonen et al. (2001) all implemented an output-based approach using unit testing to evaluate student code.

Blumenstein et al. (2004); Helmick (2007); Longo et al. (2009); Saikkonen et al. (2001) describe non-web-based systems, whereas Karavirta and Ihantola (2010); Llana et al. (2014) present web-based systems. Llana et al. (2014) evaluate student answers on the server side, whereas Karavirta and Ihantola (2010) evaluate student answers on the client side.

2.3.2. Static assessment

Static assessment assesses the student's code without running it. The goal is to find if the code fulfills some quality metrics (e.g. variable naming, comments, good programming practice).

Static assessment attempts to measure code quality, find bugs, and ensure that the student's answer is following good programming techniques. Truong, Roe, and Bancroft (2004) target fill-in-the-blank problems, where a student is asked to complete a piece of code that has missing statements. Their work aims to assess code quality, not correctness. Software metrics were used to assess the quality of the student's code. A drawback of the proposed approach is that the similarity check considers only the outline of a solution and not its details.

2.3.3. Trace-based program assessment

Trace-based assessment runs the program to make sure that certain variable values/states are changing according to the requirements. There are multiple ways of doing a trace-based assessment based on the assessment requirements. Trace-based assessment is more challenging than output-based assessment because many variations may exist in a student program that satisfies the requirements specified in the problem statement. In trace-based program assessment the code must pass first the output-based program test, then pass the trace-based test. Several systems (Gerdes, Jeuring, & Heeren, 2010; Taherkhani, Malmi, & Korhonen, 2008; Thorburn & Rowe, 1997; Xu & Chee, 2003) use trace-based program assessment. But this approach is more complex. Xu and Chee (2003) automated the diagnosis of student programs for programming tutoring systems. The main techniques presented are program standardization and semantic-level program matching. This is done to compare the student's answer to a model program.

Gerdes et al. (2010) assess student code by matching it to a model answer to ensure that the student's code is following good programming techniques. This system categorizes student code into one of four predefined categories (good, good with modifications, imperfect, and incorrect), which reflects whether it follows proper programming techniques. This system does not provide feedback to students to show the problems in their code.

The semantic code analyzer implemented in our work first does output-based assessment then static assessment to make sure that not only the output is correct but also the code written to produce this output is free of misconceptions.

3. Identifying misconceptions

We have not found any prior research that explicitly discusses student misconceptions in the context of recursion in binary trees. Karpierz and Wolfman (2014) attempt to identify student misconceptions related to binary trees, but not in recursive functions. We have conducted student interviews and analyzed student answers to questions to come up with a list of the common misconceptions for this topic.

3.1. Student interviews

Beginning with the Fall 2014 offering of a Data Structures and Algorithms (DSA) course, we have reviewed midterm exam answers. We asked students to write a recursive function that, given the root to a Binary Search Tree and a key, returns the number of nodes having key values less than K . We selected a pool of students who had answered the question incorrectly or inefficiently and requested an interview. Four students agreed to participate. We tried to determine why they solved the question incorrectly or inefficiently, and how we can help them to avoid these misconceptions. The interview questions and the transcript from the interviews can be found in Appendix A. We conclude from the interviews that the main reason for misconceptions is lack of practice with appropriate feedback that consistently warns the student about their misconceptions. That was our inspiration for building a binary tree recursion tutorial, with semantic code analysis to provide this kind of feedback.

3.2. Student exam response analysis

We analyzed student answers to test questions and student responses to an automatically assessed programming exercise on recursion in binary trees. We have analyzed more than 600 responses to binary tree recursive function writing questions given to students over three semesters (Spring 2013, Fall 2013 and Spring 2014) in the pre-test, post-test, mid-term, or final exams of a CS3 level Data Structures and Algorithms course.

Appendix C shows the questions that we have used to find student misconceptions related to recursion in binary trees. Each question rubric is tagged to the list of misconceptions and difficulties. The rubrics are shown in [Tables C1, C2, C4 and C5](#). We assume that the student solving the test should already be proficient in basic recursion as taught in the prerequisite course, so we limit our rubrics to misconceptions related to recursion in binary trees. Through analyzing student answers to recursion in binary tree questions, we found that students do express misconceptions related to recursion in binary trees more than those related to basic recursion. We counted how many times an answer

was not correct due to one of the twelve misconceptions related to basic recursion discussed in [Anonymized]. We found that roughly 30% of the wrong answers were due to problems related to understanding basic recursion, while the other 70% were due to misconceptions related to recursion in binary trees.

We have also analyzed more than 5200 attempts from 640 students in three institutions over three semesters (Spring 2013, Fall 2013 and Spring 2014) on an automatically assessed programming exercise on recursion in binary trees. The programming exercise asks the students to write a recursive function to count the number of leaf nodes in a given binary tree. For all the questions and exercises, the analysis was done manually by one of the authors who had more than eight years of teaching experience for recursion in binary trees. The instructor looked at each student answer or response and recorded the problem with the answer. Then the instructor viewed all the problems recorded to find a repeating pattern. From that, common misconceptions and difficulties encountered by students when writing functions that involve using recursion to traverse a binary tree were found.

We can identify differences between the misconceptions related to recursion in binary trees and the misconceptions that we identified for basic recursion. Most of the misconceptions related to binary trees do not necessarily lead to wrong outputs (i.e. the resulting function will pass the unit tests). Instead, many of these misconceptions tend to relate to code complexity or algorithmic inefficiency.

3.3. Common misconceptions and difficulties

The following is a list of the common misconceptions and difficulties found. We give each one an identifying tag for use in tables presented later.

- (1) In any recursive function that traverses a binary tree, we must explicitly check if the children of the current node are null or not before making the recursive call. [childsNull]

Example: Given the root to a Binary Tree, the function finds the depth of the binary tree. The depth of a binary tree is the length of the path to the deepest node. An empty tree has a depth of 0, and a tree with a root node only has a depth of 1 and so on.

```
int btDepth(BinNode root) {
    if (root == null)
        return 0;
    //Misconception:
    if (root.left() == null || root.right() == null)
        return 1;
    else {
```

```

        return 1 + Math.max(btDepth(root.left()),
        btDepth(root.right()));
    }
}

```

- (2) In any recursive function that traverses a binary tree, if we check the current node's value, then we have to check its children's values explicitly. [childCheckValue]

Example: Given the root to a Binary Search Tree (BST) and a value "key", function `bstsmallCount` returns the number of nodes having values less than key. It should visit as few nodes in the BST as possible.

```

int bstsmallCount(BinNode root , int key) {
    if (root == null)
        return 0;

    if ((Integer) root.element() < key)
        return 1 + bstsmallCount (root.left() , key)
        + bstsmallCount (root.right() , key)

    //Misconception
    if ((Integer) root.left().element() < key)
        return 1 + bstsmallCount (root.left().left() , key)
        + bstsmallCount (root.left().right() , key)
    else
        return bstsmallCount (root.left() , key);
}

```

- (3) In any recursive function that traverses a binary tree, we have to explicitly check whether the current node is a leaf or not to terminate the recursive function. [rootIsLeaf]

Example function `btDepth`:

```

int btDepth(BinNode root) {
    if (root == null)
        return 0;
    //Misconception
    if (root.isLeaf() == true)
        return 0;
    else {
        return 1 + Math.max(btDepth(root.left()),
        btDepth(root.right()));
    }
}

```

- }
 (4) In any recursive function that traverses a binary tree, we do not need to check if the root is Null. [rootIsNotNull]

Example function btDepth:

```
int btDepth(BinNode root) {
    //Missing base case to check if the root equals null
    return 1 + Math.max(btDepth(root.left()),
        btDepth(root.right()));
}
}
```

- (5) [Difficulty] In a recursive function that traverses a BST, we have to traverse the whole tree regardless of the aim of the traversal. (For example, when searching for the minimum value in the tree, the student checks the right subtree.) [BSTMinCheckRight]

Example function bstsmallCount:

```
int bstsmallCount(BinNode root , int key) {
    if (root == null)
        return 0;
    //Difficulty: missed checking the value of the key to
    traverse
    //the left tree if the key is smaller than the root
    value
    return 1 + bstsmallCount (root.left() ,key)
    + bstsmallCount (root.right() ,key)
}
}
```

- (6) [Difficulty] In a recursive function that traverses a BST, we can miss traversing critical parts of the tree. (For example, when searching for the minimum value in the tree, the student sometimes or always fails to search the left subtree.) [BSTMinNoCheckLeft]

Example: For bstsmallCount:

```
int bstsmallCount(BinNode root , int key) {
    if (root == null)
        return 0;
    //Difficulty: missed checking the left subtree
    return 1+ bstsmallCount (root.right() ,key)
}
}
```

Note that only Misconception 4 and Misconception 6 result in a function that gives a wrong answer (and then only if the initial call is made with a null tree).

However, it is essential for students to learn to avoid unnecessary code complexity. Otherwise, they will find it difficult or impossible to write more complicated recursive functions, such as operations on advanced data structures like 2–3 trees.

4. BTRecurTutor: an advanced recursion tutorial

In Hamouda, Edwards, Elmongui, Ernst, and Shaffer (2018), we described an interactive tutorial with practice exercises to teach basic recursion. Based on this experience, and realizing that even students with a good understanding of basic recursion still struggle with writing recursive methods on trees that both work and are efficient, we were inspired to develop a new tutorial aimed at teaching these more advanced recursion skills. We call this BTRecurTutor. BTRecurTutor directly attempts to overcome the misconceptions that students often have. It is a significant expansion of material originally appearing in Shaffer (2011), but which did not in its original form properly address typical student misconceptions. BTRecurTutor features programming exercises that help students to practice recursion in binary trees. A key feature of this tutorial is that we implemented semantic code analysis that detects a student's misconceptions from their answer, and provides appropriate feedback about the misconception encountered.

BTRecurTutor is presented to users as a chapter (where a chapter is defined as a series of modules) within the OpenDSA eTextbook system. Figure 1 shows BTRecurTutor as Chapter 7 in the eTextbook for a post-CS2 course on Data Structures and Algorithms (named CS3114 in the figures). Examples of modules from BTRecurTutor are shown in Figures 2 and 3.

We initially created our own platform for implementing automated assessment for programming exercises with semantic code analysis, which was used in Spring 2016. Then we migrated all of these exercises to another platform, Code Workout (Buffardi & Edwards, 2014), which did not initially support semantic code analysis; this was used in Fall 2016. More recently, we re-implemented static analysis of code support for detecting misconceptions within Code Workout, which is used currently. This is equivalent to the original system that was used by students in Spring 2016.

Figure 2 shows a tutorial module with example code at the top of the page, a visualization, and a code exercise. This exercise gives the student a problem statement and a method signature, and the student must write the method body to solve the problem. The feedback shown on the right side of the exercise is the result of running the code against test cases. This is the version that was used by Fall 2016 students. It does not do semantic code analysis. Figure 4 is another example of an exercise used in Fall 2016. This exercise appears also at the end of the Figure 3. Figure 3 shows an example of how the text and the visualizations

Chapter 7 Binary Trees

- 7.1. Binary Trees Chapter Introduction
- 7.2. Binary Trees
 - 7.2.1. Definitions and Properties
 - 7.2.2. Practice Questions
- 7.3. Binary Tree as a Recursive Data Structure
 - 7.3.1. Binary Tree as a Recursive Data Structure
- 7.4. The Full Binary Tree Theorem
- 7.5. Binary Tree Traversals
 - 7.5.1. Binary Tree Traversals
 - 7.5.1.1. Preorder Traversal
 - 7.5.1.2. Postorder Traversal
 - 7.5.1.3. Inorder Traversal
 - 7.5.1.4. Implementation
 - 7.5.2. Postorder Traversal Practice
 - 7.5.3. Inorder Traversal Practice
 - 7.5.4. Summary Questions
- 7.6. Implementing Tree Traversals
 - 7.6.1. Implementing Tree Traversals
 - 7.6.1.1. Base Case
 - 7.6.1.2. The Recursive Call
 - 7.6.2. Binary Tree Increment By One Exercise
- 7.7. Information Flow in Recursive Functions
 - 7.7.1. Information Flow in Recursive Functions
 - 7.7.1.1. Local
 - 7.7.1.2. Passing Down Information
 - 7.7.2. Binary Tree Set Depth Exercise
 - 7.7.3. Collect-and-return
 - 7.7.4. Binary Tree Check Sum Exercise
 - 7.7.5. Binary Tree Leaf Nodes Count Exercise
 - 7.7.6. Binary Tree Sum Nodes Exercise
 - 7.7.7. Combining Information Flows
 - 7.7.8. Binary Tree Check Value Exercise
 - 7.7.9. Combination Problems
 - 7.7.10. Binary Tree Height Exercise
 - 7.7.11. Binary Tree Get Difference Exercise
 - 7.7.12. Binary Tree Has Path Sum Exercise
- 7.8. Binary Tree Node Implementations
 - 7.8.1. Binary Tree Node Implementations
- 7.9. Composite-based Expression Tree
 - 7.9.1. Composite-based Expression Tree
- 7.10. Binary Tree Space Requirements
 - 7.10.1. Binary Tree Space Requirements
- 7.11. Binary Search Trees
 - 7.11.1. Binary Search Tree Definition
 - 7.11.1.1. BST Search
 - 7.11.2. BST Insert
 - 7.11.3. BST Remove
 - 7.11.4. BST Analysis
- 7.12. Dictionary Implementation Using a BST
- 7.13. Binary Tree Guided Information Flow
 - 7.13.1. Binary Tree Guided Information Flow
 - 7.13.2. Binary Search Tree Small Count Exercise
- 7.14. Multiple Binary Trees
 - 7.14.1. Mirror Image Binary Trees Exercise
 - 7.14.2. Same Binary Tree Exercise
 - 7.14.3. Structurally Identical Binary Trees Exercise
- 7.15. A Hard Information Flow Problem

Figure 1. BTRecurTutor in the CS3114 eTextbook.

```

static int ineff_count(BinNode root) {
    if (root == null) return 0; // Nothing to count
    int count = 0;
    if (root.left() != null) {
        count = 1 + ineff_count(root.left());
    }
    if (root.right() != null) {
        count = 1 + ineff_count(root.right());
    }
    if (root.left() == null && root.right() == null) {
        return 1;
    }
    return 1 + count;
}

```

When you write a recursive function that returns a value, such as counting the number of nodes in the subtree, you have to make sure that your function actually returns a value.

4 / 4 ✓

This line computes the correct value. But it does not go anywhere, as it is missing the return.

```

static int bad_count(BinNode root) {
    if (root == null) return 0; // Nothing to count
    bad_count(root.left());
    1 + bad_count(root.left()) + bad_count(root.right());
}

```

7.7.4. Binary Tree Check Sum Exercise

X286: Binary Tree Check Sum Exercise

Given a binary tree, check if the tree satisfies the property that for each node, the sum of the values of its left and right children are equal to the node's value. If a node has only one child, then the node should have the same value as that child. Leaf nodes automatically satisfy the property. Here are methods that you can use on the `BinNode` objects:

```

interface BinNode {
    public int value();
    public void setValue(int v);
    public BinNode left();
    public BinNode right();
    public boolean isLeaf();
}

```

```

1 public boolean BTchecksum(BinNode root)
2 {
3     int leftValue, rightValue;
4     leftValue=0;rightValue=0;
5     if(root == null || (root.left() == null && root.right() == null))
6         return true;
7     else

```

Behavior	Result
<input checked="" type="checkbox"/>	BTchecksum(new BinaryTree([4, 1, 3, null, null, null, 2, null, 8, null, null])).root -> false
<input checked="" type="checkbox"/>	BTchecksum(new BinaryTree([5, 3, null, null, 2, null, null])).root -> true
<input checked="" type="checkbox"/>	BTchecksum(new BinaryTree([6, 6, null, null, null])).root -> true
<input checked="" type="checkbox"/>	BTchecksum(new BinaryTree([7, null, null])).root -> true
<input checked="" type="checkbox"/>	BTchecksum(new BinaryTree([null])).root -> true
<input checked="" type="checkbox"/>	hidden
<input checked="" type="checkbox"/>	hidden

Figure 2. An example of a programming exercise in BTRecurTutor.

are presented to the student. Figure 5 shows an example of an exercise that does both output-based feedback and semantic analysis, as used in Spring 2016.

4.1. Tutorial content

The tutorial content was reviewed and refined by four instructors. Each of the instructors had more than ten years of experience in teaching recursion in binary trees. The tutorial is divided into the following modules.

- (1) Binary Tree as a Recursive Data Structure: Shows how we view a binary tree as a recursive data structure, and how that naturally leads to recursive implementations for the operations done on the binary tree.







7.13. Binary Tree Guided Information Flow

7.13.1. Binary Tree Guided Information Flow

When writing a recursive method to solve a problem that requires traversing a binary tree, we want to make sure that we are visiting the required nodes (no more and no less).

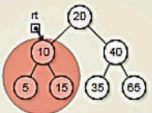
So far, we have seen several tree traversals that visited every node of the tree. We also saw the BST search, insert, and remove routines, that each go down a single path of the tree, one path through the tree. This means that the recursive function is making some decision at each node that sometimes lets it avoid visiting one or both of its children. The decision is typically based on the value of the node and the values of its children.

Here is a problem that typically needs to visit more than just a single path, but not all of the nodes.

11 / 12      

We now proceed to the recursive calls. The first recursive call counts nodes in range in the left sub-tree, and the second recursive call counts nodes in range in the right subtree.

```
int range(BSTNode root, int min, int max) {
    if (root == null)
        return 0;
    int result = 0;
    if ((min <= (Integer)root.element()) && (max >= (Integer)root.element()))
        result = result + 1;
    result += range(root.left(), min, max);
    result += range(root.right(), min, max);
    return result;
}
```



7.13.2. Binary Search Tree Small Count Exercise

X279: Binary Search Tree Small Count Exercise

Write a recursive function `BSTsmallcount` that, given a BST and a value `key`,

Figure 3. Example of a BTRecurTutor lesson (module) with an algorithm visualization (in the form of a slideshow).

- (2) Binary Tree Traversals: Shows different ways to enumerate all binary tree nodes. It covers preorder, inorder and, postorder traversals.
- (3) Implementing Tree Traversals: Shows the detailed steps needed to write a recursive function that traverses a binary tree. It covers how to write a base case, its action, a recursive call and its action, all in the context of binary tree traversals.
- (4) Information Flow in Recursive Functions: Illustrates how to handle the different types of information flow in a recursive function that traverses a binary tree. The module presents different types of information flow: local (no flow), passing down information, collect and return information upwards, and combinations of these.
- (5) Binary Search Trees: Introduces the Binary Search Tree. It also teaches (using visualizations and follow-up proficiency exercises) how to search, insert, and remove a value in a BST.
- (6) Binary Tree Guided Information Flow: Covers guided information flow, which is most relevant to operations on BSTs. It shows how when writing a recursive method to solve a problem that requires visiting a subset of

7.13.2. Binary Search Tree Small Count Exercise

X279: Binary Search Tree Small Count Exercise

Write a recursive function `BSTsmallcount` that, given a BST and a value `key`, returns the number of nodes having values less than `key`. Your function should visit as few nodes in the BST as possible.

Here are methods that you can use on the `BinNode` objects:

```
Interface BinNode {
    public int value();
    public void setValue(int v);
    public BinNode left();
    public BinNode right();
    public boolean isLeaf();
}
```

```
1 public int BSTsmallcount(BinNode root, int key)
2 {
3     if(root==null)
4         return 0;
5
6     if((Integer)root.value() < key)
7         return 1+ BSTsmallcount(root.left(), key) +
8             BSTsmallcount(root.right(), key);
9
10    else
11        return BSTsmallcount(root.left(), key);
12 }
13
```

Check my answer! Reset

Feedback

Behavior	Result
✓	BSTsmallcount((new BinaryTree({3, 2, 1, null, null, null, 4, null, 5, null, null})).root, 4) -> 3
✓	BSTsmallcount((new BinaryTree({6, 5, 4, null, null, null, 7, null, 8, null, null})).root, 4) -> 0
✓	BSTsmallcount((new BinaryTree({null})).root, 5) -> 0
✓	BSTsmallcount((new BinaryTree({2, 1, 0, null, null, null, 3, null, 4, null, null})).root, 1) -> 1

Figure 4. Example of programming exercise without semantic analyzer feedback.

X313: Binary Search Tree Small Count Exercise

Write a recursive function `BSTsmallcount` that, given a BST and a value `key`, returns the number of nodes having values less than `key`. Your function should visit as few nodes in the BST as possible.

Here are methods that you can use on the `BinNode` objects:

```
Interface BinNode {
    public int value();
    public void setValue(int v);
    public BinNode left();
    public BinNode right();
    public boolean isLeaf();
}
```

```
1 public int BSTsmallcount(BinNode root, int key)
2 {
3     if(root==null)
4         return 0;
5
6     if((Integer)root.left().value() < key)
7         return 1+ BSTsmallcount(root.left(), key)+BSTsmallcount(root.right(), key);
8
9     return BSTsmallcount(root.left(), key)+BSTsmallcount(root.right(), key);
10 }
11 }
12
```

Check my answer! Reset

Feedback

Your answer could not be processed because it contains errors:

Your solution is not efficient. There is no reason to check the children's values. Remember that the root's left or right child is treated as the new root when processed by the recursive call.

Figure 5. Example of feedback from semantic code analysis.

the nodes in a BST (such as a range query), to make sure that we are visiting the required nodes (no more and no less).

- (7) Multiple Binary Trees: Practice exercises that ask the student to implement recursive functions that perform operations on two binary trees.

- (8) **Hard Information Flow Problems:** Shows an example of how to test if a given tree is a BST. In this example, the solution is not based on purely local information, but instead depends on passing relevant information down the tree.

4.1.1. Visualizations

In order to provide explicit instruction to combat common misconceptions, we added the following visualizations to OpenDSA's binary trees chapter.

- (1) **Sum on a binary tree:** Focuses on the abstraction of recursion. This visualization uses the delegation process discussed in Edgington (2007). It shows an example of how to compute the sum of the values of all the nodes in a binary tree by delegating the task to two friends.
- (2) **BT Common Mistakes:** Shows, using code examples, the common misconceptions that students encounter when writing a recursive function that traverses a binary tree.
- (3) **BST Common Mistakes:** Traces an example of a recursive function on a BST to show the common mistakes that students display. It shows situations where one can benefit from BST properties when writing a recursive function to traverse fewer nodes.

4.1.2. Programming exercises

All programming exercises in this tutorial ask a student to write a full function that performs a certain task. [Figure 4](#) shows an example of a programming exercise. The programming exercises train the student on different types of information flows, or how to deal with multiple binary trees. The programming exercises fall into the following categories.

- (1) **Local:** This type of traversal involves performing a local operation on each node in the tree. Such tasks need no information flow between the binary tree nodes, they merely need to traverse the tree, perhaps in some particular order.
- (2) **Passing Down Information:** This type of traversal involves passing some information to nodes during the traversal process.
- (3) **Collect and return:** This type of traversal requires that we communicate information back up the tree to the caller.
- (4) **Combining Information Flows:** This type of traversal requires both that information be passed down, and that information be passed back.
- (5) **Guided:** This type of traversal should not visit every node in the tree. This means that the recursive function is making some decision at each node that sometimes allows it to avoid visiting one or both of its children. The

- decision is typically based on the value of the current node. Many problems that require information flow on BSTs are considered to be guided.
- (6) Multiple Binary Trees: This type of problem involves operations on more than one binary tree.
 - (7) Appendix B, Table B1 shows a detailed description of the exercises that are provided in BTRecurTutor.

5. Semantic code analysis

BTRecurTutor uses semantic code analysis to detect misconceptions related to recursion in binary trees in a student answer, and provides detailed feedback on the misconception that was displayed. If the student's code compiles successfully, then the infrastructure checks if the output is correct or not. If the output is correct, then the semantic code analyzer is called. The semantic analyzer is passed the student's code and the exercise name. The semantic code analyzer checks a manually predefined set of problem-specific misconceptions. If the semantic code analysis finds evidence of a misconception, then the answer is considered incorrect, and feedback from the semantic code analysis is shown to the student.

For example, if the exercise asks to find if a certain value exists in a given tree, the following will be checked by the semantic code analysis:

- (1) Does the student check if the root is null? This is required to pass all test cases successfully.
- (2) Does the student explicitly check on the child(ren) value(s)? Such checks are unnecessary for this problem, and the student should be told so.
- (3) Does the student explicitly check if the child(ren) is (are) null? Such a check is redundant for this problem due to (1), and the student should be told so.

For exercises on BSTs, the semantic code analyzer checks if the student traverses the whole tree when this is not appropriate. For example, if the aim of traversal is to find the minimum value in a BST, then the right sub-tree should not be traversed at any node.

In the post-testing phase, feedback is sent back to the client, which is then displayed to the student. Semantic code analysis is implemented for 15 binary tree programming exercises.

The semantic code analyzer can detect any of the following misconceptions:

- (1) Unnecessarily checks if the children of the current node are null or not.
- (2) Unnecessarily checks the children values whenever checking the current node's value.

- (3) Unnecessarily checks whether the current node is a leaf or not to terminate the recursive function.
- (4) Missing a check to see if the root is null.
- (5) In a recursive function that traverses a BST, process sub-trees that cannot contain nodes with the target property. For example, when searching for the minimum value in a BST, the function should not check the right subtree. Another example, when doing range query, the function should not automatically visit all children of a node.
- (6) In a recursive function that traverses a BST, miss processing sub-trees that contain nodes with the target property. For example, when searching for the minimum value in the tree, sometimes (or always) fail to search the left subtree. Or when searching for all values less than a target value, it would be wrong to never check the right child of any node.

Detailed feedback is provided to the student based on the misconception(s) displayed in her response to the programming question. [Figure 5](#) shows an example of the detailed feedback provided from the semantic code analysis.

6. Experiments

In this section, we present the results from our evaluation of the use of BTRrecurTutor in a post-CS2 DSA course, referred to as CS3114. We compare the outcomes of a control group (given without using BTRrecurTutor) against an intervention group (sections of CS3114 that used BTRrecurTutor). Specifically, we compare the outcomes on the final exams for these sections. The control groups and the experimental groups were taught by the same instructor in the same style and same teaching strategies. The instructor has more than 30 years of teaching experience. The instructor emphasised on the misconceptions for the control and experimental groups.

6.1. Control and experimental groups

The students of the control and experimental groups were students enrolled in Data Structures and Algorithms at [anonymous]. We have two control groups and two experimental groups (details shown in [Table 1](#)). The two control groups were enrolled during the semesters Fall 2011 (107 students) and Fall 2012 (57

Table 1. Control and experimental groups.

Group	N	Tutorial Status
Fall 2011 (Control 1)	107	Textual content
Fall 2012 (Control 2)	57	Textual content
Spring 2016 (Semantic Analysis)	176	15 practice exercises, semantic analysis
Fall 2016 (Plain Exercises)	192	15 practice exercises, no semantic analysis

students). Both groups had not used our binary tree tutorial and had not been assigned binary tree recursion programming exercises. The existing tutorial in those semesters had only textual content copied from the reference book used in the course (Shaffer, 2011).

The first experimental group were students enrolled during Spring 2016 (176 students) who used BTRecurTutor, including the programming exercises with the semantic analysis feature (Figure 5). The second experimental group included students enrolled in the Fall 2016 semester (192 students). This group used the binary tree tutorial with the programming exercises, but without the semantic analysis feature (Figure 4). (This loss of the semantic analysis support was due to switching from our original programming exercise system to Code Workout, which at that time did not include the semantic analysis support which checks for students misconceptions and give a feedback about it. Instead, Code Workout infrastructure just checked for the correctness of the function output against a set of test cases without checking the misconceptions in the code.) Both experimental groups were assigned all 15 programming practice exercises as graded assignments.

6.2. Evaluation question

The following question was used as an assessment for student understanding of recursion in binary trees. The question was given to the students as a part of the final exam for all the experimental and control groups. The question measures all the misconceptions related to the topic of recursion in binary trees.

Write a recursive function named `range` that, given the root to a Binary Search Tree (BST), key value `min`, and key value `max`, returns the number of nodes having key values that fall between `min` and `max`. Function `range` should visit as few nodes in the BST as possible. Function `range` should have the following prototype:

```
int range (BinNode root , Key min, Key max)
```

The evaluation question, along with the rubric showing the misconceptions covered, are shown in Table C3.

6.3. Results

During Fall 2011, Fall 2012, Spring 2016, and Fall 2016, students were given the same evaluation question on the final exam shown in Table C3. Course teaching assistants graded the questions to a specific rubric that includes the items shown in Table C3, with point deductions specified for each item. Care was taken to keep grading consistent both within and across semesters.

Below, we have compared the student scores on the question between the following pairs: Fall 2011 versus Fall 2012, Fall 2011 versus Spring 2016, Fall 2012

Table 2. A t-test comparing the scores of the binary tree recursion exam question for control 1 (N = 107) versus control 2 (N = 57).

	Control 1		Control 2		p-value
	mean	std dev.	mean	std dev.	
score	13.51	5.54	14.28	5.6	0.401

* = statistically significant

versus Spring 2016, Fall 2011 versus Fall 2016, Fall 2012 versus Fall 2016, and Spring 2016 versus Fall 2016.

Table 2 shows the results of the unpaired t-test comparing the student scores for Fall 2011 (control) versus Fall 2012 (control). These two groups were given the material in traditional textbook style: as prose, code snippets, and still images. Practice for both groups consisted of a single paper homework exercise to write a recursive tree traversal, similar to the exam question. The t-test shows no statistically significant difference between the groups. This is the expected result, indicating that different groups of students taking this class, when using a traditional textbook, tend to get the same exam scores.

Table 3 shows the results of the unpaired t-test comparing the scores of the binary tree recursion exam question for Control 1 (N = 107) versus Semantic Analysis (N = 176).

Table 4 shows the results of the unpaired t-test comparing the scores of the binary tree recursion exam question for Control 2 (N = 57) versus Semantic Analysis (N = 176).

The results show a statistically significant improvement in the performance of the experimental group ($p = 0.0001$).

Tables 5 and **6** show the results of the unpaired t-test comparing the student scores for Fall 2011 and Fall 2012 (control) versus Fall 2016 (the experimental group that had the tutorial with 15 programming exercises, but without semantic analysis). The results show a statistically significant improvement in the performance of the experimental group ($p = 0.0001$).

Table 3. A t-test comparing the scores of the binary tree recursion exam question for control 1 (N = 107) versus semantic analysis (N = 176).

	Control 1		Semantic Analysis		p-value
	mean	std dev.	mean	std dev.	
score	13.51	5.54	18.81	3.07	0.0001*

* = statistically significant

Table 4. A t-test comparing the scores of the binary tree recursion exam question for control 2 (N = 57) versus semantic analysis (N = 176).

	Control 2		Semantic Analysis		p-value
	mean	std dev.	mean	std dev.	
score	14.28	5.60	18.81	3.07	0.0001*

* = statistically significant

Table 5. A t-test comparing the scores of the binary tree recursion exam question for control 1 (N =107) versus plain exercises (N =192).

	Control 1		Plain Exercises		p-value
	mean	std dev.	mean	std dev.	
score	13.51	5.54	16.39	4.77	0.0001*

* = statistically significant

Table 6. A t-test comparing the scores of the binary tree recursion exam question for control 2 (N = 57) versus plain exercises (N =192).

	Control 2		Plain Exercises		p-value
	mean	std dev.	mean	std dev.	
score	14.28	5.6	16.39	4.77	0.005*

* = statistically significant

Table 7. A t-test comparing the scores of the binary tree recursion exam question for semantic analysis (N = 176) versus plain exercises (N =192).

	Semantic Analysis		Plain Exercises		p-value
	mean	std dev.	mean	std dev.	
score	18.81	3.07	16.39	4.77	0.0001*

* = statistically significant

These results are consistent with improved performance by students who used BTRecurTutor.

Table 7 shows the results of the unpaired t-test comparing the student's scores for Spring 2016 which is the experimental group who had the tutorial with 15 programming exercises with semantic analysis, versus Fall 2016 which is the experimental group who had the tutorial with 15 programming exercises, but without semantic analysis. The results show a statistically significant improvement in the performance of the experimental group that used the tutorial with the semantic analysis feature ($p = 0.0001$).

Table 8 summarizes all the results and the effect sizes. In Valentine and Cooper (2003) and Cohen (2013), an effect size labelled small if it is below 0.20. It was suggested that a large effect size when it is above 0.80. Medium-sized effects have values between 0.20 and 0.80. We found a large effect size when comparing the experimental group who practiced the exercises that supports semantic code analysis to the control groups. We found a medium effect size when comparing the group who practiced the semantic code analyzed exercises to the group who practiced the exercises that does not

Table 8. Summary of the results.

Groups	Effect Size	Result
Control 1 vs Control 2	N/A	Not statistically significant
Semantic Analysis vs Controls	1.1	Statistically significant
Plain Exercises vs Controls	0.5	Statistically significant
Semantic Analysis vs Plain Exercises	0.6	Statistically significant

support semantic code analysis. We found a medium effect size when comparing the group who practiced the exercises that does not support semantic code analysis to the control groups. These results are consistent with improved performance by students who used BTRecurTutor with programming exercises and even better performance when used the programming exercises with semantic code analysis that warns them about their misconceptions.

Figure 6 summarizes the means for the Binary Tree question exam grade across years.

6.4. Threats to validity

There are many factors that might be considered as threats to validity. We attempt to address them in this section.

- (1) **Instructor:** The control groups and the experimental groups were taught by the same instructor in the same style and same teaching strategies. This instructor does not teach the course every semester, and other instructors use different exams. That is why the control and experimental groups were 5 years apart. It is worth noting that teacher's skills may well change during any five-year period.

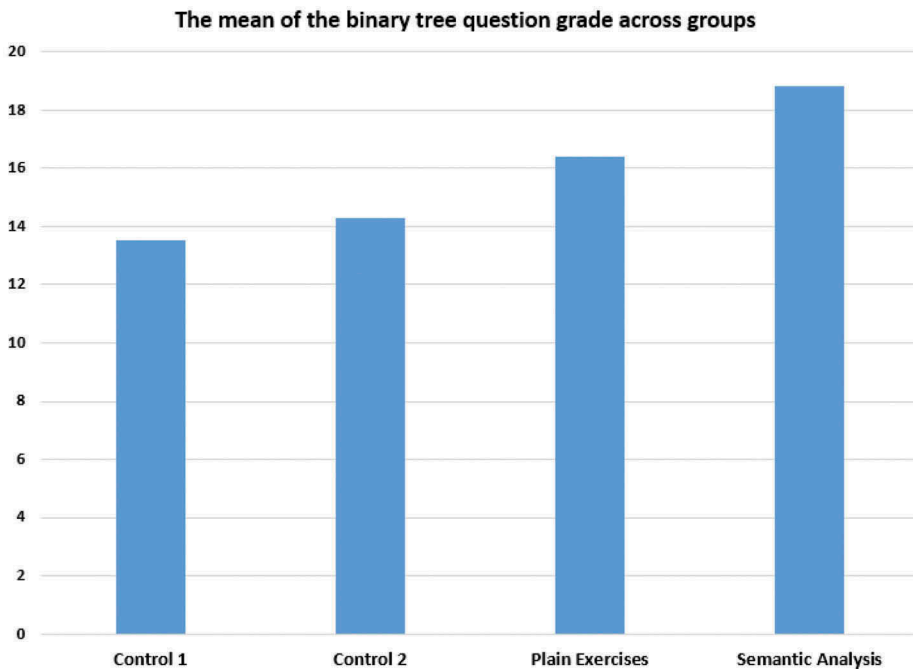


Figure 6. Binary tree exam question grade across years.

- (2) **Material used for teaching:** There was improvement in both the textbook materials presentation (the tutorial), and also in the classroom presentation to conform more closely to the tutorial content and the lessons learned over time about misconceptions. An important finding of our work is that a focus on misconceptions along with practice improves the outcomes from the instruction. Another important and independent finding is that giving better feedback on the quality of the student solutions (the semantic analysis) provides additional improvements in performance. The primary difference was the textbook/tutorial presentation with exercises for practice.
- (3) **Assessment:** The exam given to the students in the control and experimental groups were both paper-based exams where students cannot get feedback from a compiler. This might make it harder to answer questions, but both groups used this approach on the exam.
- (4) **Grading:** When grading the exams, the graders were unaware of the ongoing research study. A detailed rubric including the items shown in Table C.3 was used, with specific point deductions specified. Cross-grader variation is always possible, but does not appear likely in this relatively controlled circumstance. In addition, final exams are not returned to students, and the control and experimental groups were not taking the exam in successive semesters. These facts both minimize the concern that scores improved across years because students knew about the questions in advance.
- (5) **Time-on-task:** Nearly all students in the experimental groups completed all exercises. These are given for homework credit, within the OpenDSA framework. It uses a mastery-based approach, meaning that students can repeat the exercises until they get them correct. As a result, few students do not complete the exercises. This means that students in the experimental group had more time practicing coding for binary trees, which might be one of the factors enhancing the grades. In addition, adding the semantic analyzer to the exercises provided a better feedback to the students to understand what is the misconception they have in their answers. In the version of exercises where no semantic analysis supported the feedback is based on if the code output is correct or wrong. The semantic code analyzer added quality to the time spent practicing. In other words, the fact that students on their own will take insufficient time to practice has itself been determined to be a problem with typical instructional practice on recursion, and our tutorial helps to overcome this problem by explicitly requiring this needed practice. We found in prior work (Hamouda et al., 2018) that practicing basic recursion exercises helps with learning basic recursion topics. As a side effect of its design and delivery, students did spend more time with BTRecurTutor than previous students spent with only textual content presented to them.

Since our previous surveys of instructors (Hamouda et al., 2018) indicates that they believe students do not spend enough time practicing recursion, we view this as generally a positive outcome.

7. Conclusion

This paper presents our efforts to enhance the learning of recursion in binary trees as it is typically taught post CS2. This is often cast in the form of recursive operations on binary trees. First, we have identified the misconceptions that students have in understanding recursion in binary trees through the analysis of answers of recursion in binary tree questions and student interviews. Then, we have designed a question to measure those misconceptions. We have designed a rubric to match possible answers to misconceptions. Finally, we built a tutorial that addresses those misconceptions and trains students to avoid them. The tutorial features code writing questions. It trains the students in part through feedback from a semantic code analyzer that detects misconceptions in the students' answers to the practice exercises. Our results show an enhancement in student performance when using the tutorial with the practice exercises (but no semantic analysis feedback), and even more enhancement when using the same exercises with appropriate feedback about the misconceptions detected in the answer provided.

We conclude that the best way to use BTRecurTutor to enhance student performance on recursion is to allow students to practice advanced recursion by solving the tutorial exercises. The results suggest that additional practice with the tool improves performance. Beyond additional practice, the semantic feedback enhances student performance even further. However, further analysis is needed to understand what aspects of the practice exercises on BTRecurTutor leads to the enhancement in student performance. There are two distinct aspects of BTRecurTutor that might affect the learning of recursion. One is that BTRecurTutor explicitly delivers instruction aimed at teaching advanced recursion and overcoming misconceptions. The other is that BTRecurTutor involves the extensive practice of advanced recursive skills, with writing recursive functions. Our design is unable to distinguish the relationships between these effects. However, given that providing feedback based on semantic analysis of student's exercise solutions results in a significant boost in improvement, it seems likely that practice (even without feedback beyond correctness) is likely to give a performance boost.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by the National Science Foundation [DUE-0836940, DUE-0937863, and DUE-0840719];VT-MENA.

Notes on contributors

Sally Hamouda is an Assistant Professor of Computer Science at Rhode Island College. She received her PhD from Virginia Tech. Dr. Hamouda's research area are Computer Science Education and Data mining. She is interested in discovering student misconceptions in different hard topics in CS and how to resolve them.

Stephen H. Edwards is a Professor and the Associate Department Head for Undergraduate Studies in the Department of Computer Science at Virginia Tech, where he has been teaching since 1996. He received his B.S. in electrical engineering from Caltech, and M.S. and Ph.D. degrees in computer and information science from The Ohio State University. His research interests include computer science education, software testing, software engineering, and programming languages. He is the project lead for Web-CAT, the most widely used open-source automated grading system in the world. Web-CAT is known for allowing instructors to grade students based on how well they test their own code. In addition, his research group has produced a number of other open-source tools used in classrooms at many other institutions.

Hicham G. Elmongui is an Associate Professor, Department of Computer and Systems Engineering, Faculty of Engineering, Alexandria University. His research interests lie in the area of data engineering. Specifically, interested in query processing and optimization. Prof. Elmongui is interested in security engineering and the embedding of security business rules into the software development life cycle.

Jeremy V. Ernst is a Professor and Associate Dean for Research at Embry-Riddle University. He specializes in research focused on dynamic intervention means for STEM education students categorized as at-risk of dropping out of school. Dr. Ernst has teaching, advising, and research experiences in various capacities at the postsecondary and secondary levels.

Clifford A. Shaffer is an Associate Department Head for Graduate Studies and Professor of Computer Science at Virginia Tech. where he has been since 1987. He received his PhD from University of Maryland in 1986. Over his career, Dr. Shaffer's research efforts have spanned three major themes: Data structures and algorithms for spatial applications, integrated problem-solving environments for engineering and science applications (most notably for systems biology), and simulation and visualization for education (including Computer Science, Statistics, and Geography). He has been PI or Co-PI for over \$10,000,000 in research funding. Dr. Shaffer has published nearly 200 journal and conference papers.

ORCID

Hicham G. Elmongui  <http://orcid.org/0000-0001-5947-7450>

References

- Adams, W. K., & Wieman, C. E. (2011). Development and validation of instruments to measure learning of expert-like thinking. *International Journal of Science Education*, 33(9), 1289–1312.
- Blumenstein, M., Green, S., Nguyen, A., & Muthukkumarasamy, V. (2004, June). An experimental analysis of GAME: A generic automated marking environment. *SIGCSE Bulletin*, 36(3), 67–71.
- Buffardi, K., & Edwards, S. (2014). Introducing codeworkout: An adaptive and social learning environment. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education, (SIGCSE'14)* (pp. 724). Atlanta, Georgia.
- Chaffin, A., Doran, K., Hicks, D., & Barnes, T. (2009). Experimental evaluation of teaching recursion in a video game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, (pp. 79–86). New Orleans, Louisiana.
- Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Routledge, New York University, New York.
- Dalkey, N., & Helmer, O. (1963). An experimental application of the Delphi method to the use of experts. *Management Science*, 9(3), 458–467.
- Danielsiek, H., Paul, W., & Vahrenhold, J. (2012). Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd acm technical symposium on computer science education*, (pp. 21–26). Raleigh, North Carolina.
- Edgington, J. (2007, October). Teaching and viewing recursion as delegation. *Journal of Computing Sciences in the Colleges*, 23(1), 241–246.
- Gerdes, A., Jeuring, J., & Heeren, B. (2010, November). Using strategies for assessment of programming exercises. *SIGCSE Bulletin*, 40(4), 441–445.
- Ginat, D., & Shifroni, E. (1999). Teaching recursion in a procedural environment-how much should we emphasize the computing model? In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education, (SIGCSE'99)* (pp. 127–131). New Orleans, Louisiana.
- Grissom, S., Murphy, L., McCauley, R., & Fitzgerald, S. (2016). Paper vs. computer-based exams: A study of errors in recursive binary tree algorithms. In *Proceedings of the 47th acm technical symposium on computing science education*, (pp. 6–11). Memphis, Tennessee.
- Hamouda, S., Edwards, S., Elmongui, H., Ernst, J., & Shaffer, C. (2017). A basic recursion concept inventory. *Computer Science Education*, 27(2), 121–148.
- Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2018, November). Recurtutor: An interactive tutorial for learning recursion. *ACM Transaction Computing Education*, 19(1), 1:1–1: 25. Retrieved from
- Helmick, M. (2007). Interface-based programming assignments and automatic grading of java programs. In *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education, (ITiCSE'07)* (p. 63–67). Dundee, Scotland.
- Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st acm technical symposium on computer science education*, (pp. 107–111). Milwaukee Wisconsin.
- Karavirta, V., & Ihanntola, P. (2010). Serverless automatic assessment of Javascript exercises. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education, (ITiCSE'10)* (p. 303). Ankara Turkey.
- Karpierz, K., & Wolfman, S. (2014). Misconceptions and concept inventory questions for binary search trees and hash tables. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education, (SIGCSE'14)* (pp. 109–114). Atlanta Georgia.

- Laakso, M., Salakoski, T., & Korhonen, A. (2005). The feasibility of automatic assessment and feedback. *International Conference on Cognition and Exploratory Learning in Digital Age*, (pp. 113–122). Porto, Portugal.
- Llana, L., Martin-Martin, E., Pareja-Flores, C., & Velázquez-Iturbide, J. (2014). FLOP: A user-friendly system for automated program assessment. *Journal of Universal Computer Science*, 20(9), 1304–1326.
- Longo, P., Sterbini, A., & Temperini, M. (2009). *TSW: A web-based automatic correction system for C programming exercises*. Berlin, Heidelberg: Springer.
- Murphy, L., Fitzgerald, S., Grissom, S., & McCauley, R. (2015). Bug infestation!: A goal-plan analysis of CS2 students' recursive binary tree solutions. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, (pp. 482–487). Kansas City, Missouri.
- Paul, W., & Vahrenhold, J. (2013). Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, (pp. 29–34). Denver, Colorado.
- Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of oop concepts by novices. *Computer Science Education*, 15(3), 203–221.
- Rowe, G., & Smaill, C. (2007). Development of an electromagnetic courseconcept inventorya work in progress. *Proc. 18th conf. australian association for engineering (aace)*, Melbourne, Australia.
- Saikkonen, R., Malmi, L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *Proceedings of the Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*, (pp. 133–136). Canterbury, United Kingdom
- Scholtz, T., & Sanders, I. (2010). Mental models of recursion: Investigating students' understanding of recursion. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)*, (pp. 103–107). Ankara, Turkey.
- Shaffer, C. (2011). *Data structures and algorithm analysis in java* (3rd ed.). Pearson: Dover Publications.
- Taherkhani, A., Malmi, L., & Korhonen, A. (2008). Algorithm recognition by static analysis and its application in students' submissions assessment. *Proceedings of the Eighth International Conference on Computing Education Research (Koli'08)*, (pp. 88–91). Koli, Finland.
- Taylor, C., Zingaro, D., Porter, L., Webb, K., Lee, C., & Clancy, M. (2014). Computer science concept inventories: Past and future. *Computer Science Education*, 24(4), 253–276.
- Tessler, J., Beth, B., & Lin, C. (2013). Using cargo-bot to provide contextualized learning of recursion. *Proceedings of the Ninth Annual ACM Conference on International Computing Education Research (ICER'13)*, (pp. 161–168). San Diego, California .
- Thorburn, G., & Rowe, G. (1997, December). PASS: An automated system for program assessment. *Computer Education*, 29(4), 195–206.
- Truong, N., Roe, P., & Bancroft, P. (2004). Static analysis of students' java programs. *Proceedings of the Sixth Australasian Conference on Computing Education*, (pp. 317–325). Darlinghurst, Australia.
- Valentine, J. C., & Cooper, H. (2003). *Effect size substantive interpretation guidelines: Issues in the interpretation of effect sizes* (pp. 1–7). Washington, DC: What Works Clearinghouse.
- Wilcocks, D., & Sanders, I. (1994). Animating recursion as an aid to instruction. *Computers and Education*, 23(3), 221–226.
- Xu, S., & Chee, Y. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4), 360–384.
- Zingaro, D., Petersen, A., & Craig, M. (2012). Stepping up to integrative questions on CS1 exams. *Proceedings of the 43rd acm technical symposium on computer science education*, (pp. 253–258). Raleigh, North Carolina. doi:10.1145/2157136.2157215

Appendix A. CS 3114 Interviews

A.1. The interview questions

- (1) How confident are you about writing recursive functions?
- (2) How confident are you about writing recursive programs related to binary trees and traversals?
- (3) What was the reason for the wrong answer on the recursion in binary tree mid-term Question?
- (4) Do you think that you have learned more about writing recursive functions since you took the midterm?
- (5) Do you think that you could now write this function correctly?
 - (a) If yes, how would you figure out a fix?
 - (b) If no, why could not you figure out a fix?
- (6) What do you think could help you to understand the topic of recursive tree functions better?
- (7) Do you have any suggestions on enhancing the presentation of the binary trees chapter in OpenDSA?

A.2. Subject Responses

Subject 1

The student usually uses OpenDSA to prepare for the midterms by reading the chapter and then re-reading it one or two days before the midterm.

He thinks that understanding data structures is harder than understanding sorting.

He is generally comfortable with recursion. We asked how he would solve the programming exercise to count the leaf nodes in a given tree. The student could not remember how he approached it.

Reviewing his progress from tracking data stored in our database, we found that he had made three attempts until he got the correct answer. However, his answer did unnecessary checks that could be avoided. In every attempt, he tried to modify his code so that he can fix the errors that appeared to him. His answer to the OpenDSA programming exercise shows that he is not good at formulating the recursive case, nor the recursive call.

He reported that OpenDSA exercises about binary trees are too easy. He thinks that the presentation of binary trees in OpenDSA should be enhanced by showing real applications. He suggested that OpenDSA can have more difficult code-writing exercises on binary trees and traversals.

After showing him his mid-term answer, he reported that the main reason for not getting the correct answer for the binary tree exercise is not writing the base case correctly. However, he was not sure how to fix his answer. He did not have any suggestions about how he could enhance his understanding of binary trees, because he says that he did not study it well enough.

Subject 2

She used OpenDSA to study for the mid-terms and used some of the OpenDSA examples, definitions, data structures usages in her “cheat sheet” that students were allowed to bring to the midterms. This subject had a year gap between taking CS2114 and CS3114 for a reason related to a family emergency, and that is why she used to work alone in the projects. She thinks that gap affected her performance in 3114, especially in her programming skills. Her

first language is not English, that is why she thinks that she spends more time than her classmates in reading and understanding the material and the terminologies. She said that CS2114 is not doing an excellent job in preparing students for CS3114, and there is a gap between them. She said that recursion is not well covered in CS2114, and that is why she struggled in CS3114 on the topics that are related to recursion.

Her confidence level about writing recursive functions is 4 on a scale from 1 to 10. She said that the practice exercises on trees and traversals are too easy. Our collected data show that she made 15 attempts before getting the answer correct in the programming exercise. She said she depended on the feedback she got from the programming exercise editor in OpenDSA to fix her errors, and also she has looked on the internet to find out how to fix her problems.

On reviewing her midterm answer to the programming exercise, she could not figure out why her answer was wrong and how to fix it. She said she learned more about recursion after the first midterm. She suggested that OpenDSA should have more practice and visualizations for the tree traversal topic to help the student understand it better.

Subject 3

The subject used OpenDSA and Google to study for the mid-terms. He generally likes to have multiple sources to study from. He is pretty confident in writing general recursive functions and recursive functions on trees. He thinks he is not a good test taker and that is why he missed the answer to the binary trees programming exercise on the mid-term, as he left that question for the last 5 minutes. On reviewing his midterm answer, he could figure out his errors. He reports that recursion is easy. He thinks that adding more visualizations that show how a recursive function works on some examples will be beneficial in understanding recursion. He suggests also adding more programming exercises with different styles and tasks. He said that he uses a stack to trace how a recursive function works. He generally avoids using recursion on problems that can be solved in another way. He has two years of programming experience, and he thinks that OpenDSA helps him to understand how recursion works. He says that OpenDSA is an excellent system to learn from. He found that he had a problem in binary tree traversals in the recursion pre-test and mid-term although he thinks that he does not have any problems. He made five attempts on the OpenDSA programming exercise before getting the correct answer.

Subject 4

He uses OpenDSA to prepare for the mid-term. He uses Google as a studying resource, but not for preparing for the mid-term. He reports that OpenDSA helped him a lot. He reports that his understanding of recursion was enhanced by the CS3114 in-class explanations. He got from the class that in order to understand recursion, you need to understand well how the base case is working and avoid thinking about the details of the recursion. He thinks that recursion was made much more straightforward in CS3114. He is confident in writing recursive functions. When he was given his mid-term question, he was able to fix it quickly as he had learned more about recursion after that mid-term. He thinks that the current programming exercise on binary tree traversals is too easy and the reason why he made eight attempts is that he misread it and then when he read it correctly it worked just fine. He thinks that adding more complicated programming exercises would help better understanding of binary tree traversals.

He feels comfortable about recursion. He mentioned that the trigger to understanding was the explanation of recursion in class. In particular, he cited focusing on base cases and simplifying what he pays attention to in writing the recursive function (not looking at too many nodes).

Appendix B. BTRecurTutor Practice Exercises Detailed Information

Table B1. Writing practice exercises detailed information in recursion in binary trees tutorial.

Exercise Name	Description	Category
Increment	Increment all the values of the nodes of a binary tree by one	Local
Count Leaf	Count the number of Leaf Nodes in a binary Tree	Collect and return
Depth	Get the depth of a binary Tree	Collect and return
Check Value	Check on the existence of a given value in a binary tree	Collect and return
Count Value	Count the number of existences of a given value in a binary tree	Collect and return
Sum All	Sum all the values of the nodes in a binary tree	Collect and return
Has Path Sum	Check if any path from the root to a leaf has a given sum	Collect and return
Get Difference	Get the difference between the values on the left sub-tree and the right sub-tree	Collect and return
Diameter	Get the diameter of a given sub-tree	Collect and return
CheckSum	Check if for each node if the sum of its children is equals to its value	Collect and return
Minimum	Find the minimum in a binary search tree	Guided
Small Count	Count the number of existences of a node value less than a given value in a binary search tree	Guided
Same Tree	Check if the values in two given trees are the same	Multiple Trees
Swaps Trees	Swap the values of two given trees	Multiple Trees
Structurally Identical Trees	Check if two given trees are structurally identical	Multiple Trees
Mirror Trees	Check if the values in the nodes of two given trees are mirrored	Multiple Trees

Appendix C. Test Questions

C.1. Pre-test Questions

- Write a recursive function named `bstMin` that, given the root to a Binary Search Tree (BST), returns a reference to the node that has the minimum value found in the passed tree. Function `bstMin` should visit as few nodes in the BST as possible. Function `bstMin` should have the following prototype:

```
BinNode bstMin(BinNode root)
```

The Correct Answer :

```
BinNode bstMin(BinNode root) {
    if (root == null)
        return null;
    if (root.left() == null)
        return root;
    return bstMin(root.left());
}
```

Table C1. Question item 1. Rubric.

Answer	Misconception
Solution that access <code>root.right()</code>	BSTMinCheckRight
Solution that does not access <code>root.left()</code>	BSTMinNoCheckLeft
Solution that does not check if <code>root == null</code>	RootIsNotNull
Solution that checks on <code>root.isLeaf()</code> as the base case	RootIsLeaf
Other	?

2. Write a recursive function named `btCheckVal` that, given the root to a Binary Tree and value, returns true if there is a node in the given binary tree with the given value, and false otherwise. Function `btCheckVal` should have the following prototype:

```
boolean btCheckVal (BinNode root , int value)
```

The Answer :

```
boolean btCheckVal (BinNode root, int value) {
    if (root == null)
        return false;
    else {
        if (root.element() == value)
            return true;
        else
            return btCheckVal (root.left(), value) ||
                btCheckVal (root.right(), value);
    }
}
```

Table C2. Question item 2. Rubric.

Answer	Misconception
Solution that accesses the values of the root's left or right children	ChildCheckValue
Solution that checks if the root's left or right children is null	ChildIsNull
Solution that does not check if root == null	RootIsNotNull
Solution that checks on root.isLeaf() as the base case	RootIsLeaf
Other	?

C.2. Post-test Questions

- (3) Write a recursive function named `bstsmallCount` that, given the root to a Binary Search Tree (BST) and a value "key" returns the number of nodes having values less than key. Function `bstsmallCount` should visit as few nodes in the BST as possible. Function `bstsmallCount` should have the following prototype:

```
int bstsmallCount (BinNode root , int key)
```

The Answer :

```
int bstsmallCount (BinNode root , int key) {
    if (root == null)
        return 0;
    if ((Integer) root.element() < key)
        return 1 + bstsmallCount (root.left(), key)
            + bstsmallCount (root.right(), key)
    else
        return bstsmallCount (root.left(), key);
}
```


Table C3. Question item 3. Rubric.

Answer	Misconception
Solution that access root.right() in the else condition	BSTMinCheckRight
Solution that does not access root.left()	BSTMinNoCheckLeft
Solution that does not check if root == null	RootIsNotNull
Solution that checks on root.isLeaf() as the base case	RootIsLeaf
Solution that access the values of the root's left or right children	ChildCheckValue
Solution that check if the root's left or right children is null	ChildIsNull
Other	?

- (4) Write a recursive function named `btDepth` that, given the root, to a Binary Tree the function finds the depth of the binary tree. The depth of a binary tree is the length of the path to the deepest node. An empty tree has a depth of 0, and a tree with a root node only has a depth of 1 and so on. Function `btDepth` should have the following prototype:

```
int btDepth (BinNode root)
```

The Answer:

```
int btDepth (BinNode root) {
    if (root == null)
        return 0;
    else {
        return 1 + Math.max(btDepth (root.left()),
            btDepth (root.right()));
    }
}
```

Table C4. Question item 4. Rubric.

Answer	Misconception
Solution that check if the root's left or right children is null	ChildIsNull
Solution that does not check if root == null	RootIsNotNull
Solution that checks on root.isLeaf() as the base case	RootIsLeaf
Solution that misses a recursive call on the root.left() or the root.right()	?
Other	?

C.3. Evaluation Question

- (5) Write a recursive function named `range` that, given the root to a Binary Search Tree (BST), key value `min`, and key value `max`, returns the number of nodes having key values that fall between `min` and `max`. Function `range` should visit as few nodes in the BST as possible. Function `range` should have the following prototype:

```
int range (BinNode root, Key min, Key max)
```

The Answer:

```
int range (BSTNode root, int min, int max) {
    if (root == null)
```

```

    return 0;
int result = 0;
if ((min ≤ (Integer)root.element()) &&
    (max ≥ (Integer)root.element()))
    result = result + 1;
if (min ≤ (Integer)root.element())
    result += range(root.left(), min, max);
if (max > (Integer)root.element())
    result += range(root.right(), min, max);
return result;
}

```

Table C5. Question item 5. Rubric.

Answer	Misconception	Deductions
Solutions that access root.right() unconditionally	BSTMinCheckRight	-8
Solutions that access root.left() unconditionally	BSTMinNoCheckLeft	-8
Solutions that do not check if root == null	RootIsNotNull	-2
Solutions that check on root.isLeaf() as the base case	RootIsLeaf	-2
Solutions that access the values of the root's left or right children	ChildCheckValue	-8
Solutions that check if the root's left or right children are null	ChildIsNull	-4