

Using Counter-Examples in the Data Structures Course

Alan Fekete
School of Information Technologies
University of Sydney 2006
Australia
fekete@it.usyd.edu.au

Abstract

In many computer science courses, students face a sequence of activities that is quite monotonous in style: most assignments and labs ask the student to write a program that does something, to follow a procedure, or else to prove something. We advocate varying the pattern by including different types of activity which bring students to reflect on their ideas and especially to confront directly common misconceptions. We use the traditional “data structures” subject as the domain in which to present this approach to teaching. We give detailed accounts of several activities that include debugging wrong code and explaining the misconception in an argument.

Keywords: Data structures, misconceptions, counter-examples, closed laboratories, metacognition.

1 Introduction

The coverage of data structures has been a core topic in Computer Science curriculum for decades. This course was called CS2 in the ACM curriculum of 1978, and now forms CS103O, CS103I, or CS112I in the various sample structures within CC2001 [0]; its content has changed very little since 1968 when Knuth [12] wrote his classic account of the structures and algorithms for manipulating linked lists, binary search trees, and linear probing hash tables (the only major innovations are that B-tree is now standard material, while the multi-linked storage used in Coda-syl databases has disappeared from teaching). The framework in which these structures are presented has changed a bit more: the recent influence of object-oriented languages has led to the Abstract Data Type being expressed explicitly in code (as an interface in Java, say) where it used to be merely a concept in the exposition, and also to use of Design Patterns as a theme in the course [7,13,14,16]

A student who studies data structures is taught a substantial amount of quite technical material. In this paper, however, our focus is not on the content but rather on the type of activities the student engages in. If we look at the exercises in different textbooks, we see that they are dominated by the following

- the student is asked to write code to perform an operation on a data structure

- the student is asked to show the change in a data structure resulting from an operation (in essence, this asks the student to follow a procedure that has been described)
- the student is asked to prove a property of a data structure
- the student is asked to determine the cost of an operation

For example, let’s look at the chapter on search trees in several standard textbooks. In chapter 19 in the textbook by Weiss [17], there are 6 short exercises, which all involve following a procedure (eg insert a given sequence of elements into an empty BST) or else drawing all trees containing a given set of entries. There are 8 “theory” exercises asking for proof of some property, or design of an algorithm to perform a task. There are 8 “practice” exercises and 10 “projects” all of them asking for code to be written that acts in a given way. In chapter 10 of Preiss [15] there are 20 exercises of which 5 ask the student to follow a procedure, 2 ask for a proof of a property, 3 ask for the calculation of a quantity such as running time or height of a tree, 2 ask the student to rewrite recursive code iteratively, and 7 call for the student to invent an algorithm to perform a task. The textbook by Goodrich and Tomassia [10] is the most varied, with chapter 9 containing questions including some that ask a student to decide whether a statement is true or false, some that ask the student to construct examples with given properties, some that require finding ways to model one structure in terms of another, and there are even 3 questions that ask “what is wrong with a certain argument”. Nevertheless, of 21 “reinforcement” exercises, 9 ask for a procedure to be followed, and of 25 “creativity” exercises, 17 ask for the invention of an algorithm for a given task, and 6 ask the student to prove a property.

What learning is provoked in the student when carrying out these exercises? Perhaps the most obvious benefit is *reinforcement*. For a student who knows the material, practice makes it faster, and more reliable, needing less close thought; this is especially important for skills that will later be needed as steps within a larger procedure (for example, rapid and accurate analysis of algorithms is needed as a step when designing sensible algorithms). Educational theory suggests that there is also benefit for a student in making conceptual connections between related material. Thus an activity which takes an algorithmic idea (pseudocode) and elaborates this to the detail needed for a program, will lead to a stronger mastery of the concept itself. There is also a possibility that practical activity may provoke *engagement*: many students ignore the material they see in lectures, but they will

go back and study it closely when they are required to use it in a lab or assignment.

However, all the benefits described above occur when students succeed with the assigned task. Universal experience is that many students hand in incorrect work, and they seem unaware that it is wrong. Thus the assignment has not led to correct learning. Instead these students are in a state of *misconception*, that is, they have developed a way of thinking about the topic, and are confident that they understand, but their idea is incorrect. It is conceivable that getting back an assignment with a low mark may lead a student to revisit the topic, and correct their misconceptions. However my observations suggest that many students persist in their misconception; some simply accept a low mark as a sign that they didn't put their ideas into practice properly, while others try to find a small "tweak" to their technique that will cover the case that arose in the recent work. In programming classes this is often seen as "programming by random walk".

Recently, misconceptions have attracted much attention in Computer Science Education. Recursion has been a particular focus [3,8,9], and Holland et al [11] identified misconceptions in the object concept. Several researchers have followed constructivist theory in using phenomenological research [2] to try to identify common student misconceptions.

Our paper looks at the impact on education after identifying the misconceptions. We ask: "once we have identified a common misconception, how can we adapt our teaching to deal with it?". Following Holland et al [11], we propose to provide students with explicit examples of incorrect work which was produced because of a misconception. We go significantly beyond the simple idea of [11], that students should observe the error, and we ask the student to actively identify the problem, correct it, and explain why the mistake was made. When students think about their own learning and understanding, this is called *meta-cognition*, and we have used it in other subjects to encourage deep approaches to learning [6]. We refer to examples of incorrect work as *counter-examples*, somewhat modifying the mathematical tradition where the term is used for an correct example that proves that a conjecture is false.

In Section 2, we offer a detailed overview of the various ways one can use a counter-example to provoke learning. In the remaining sections, we look at some specific case studies, each addressed to a common misconception among students in a data structures class. These examples come from our experience in a large public university, where the Data Structures course is the third semester in a Computer Science sequence. It follows two semesters of programming, which use an "objects-early" approach and a strong focus on substantial group projects. The first two semesters cover inheritance, recursion, asymptotic analysis of simple looping code, recursive descent parsing, file I/O, and the storage of elements in collections (a simple sequence, and a simple key-value mapping). The first two semesters are taught using Problem-Based Learning, and are described in earlier papers [1,5]. The data structures subject is taught conventionally, and class contact is made up of lectures and a two-hour closed lab each week (each 20 students has a tutor, usually a third or fourth year undergraduate, who is present for one of the lab hours only.) There have been about 550 students taking the subject each year, made up of approximately 250 intending CS majors, 200 Electrical Engineering majors, and a diverse group with other interests such as business, math, physics, even humanities. The class

uses Java as the programming language, so that will be the vehicle for all the code fragments we present.

2 Counter-examples in teaching

Of course one should use counter-examples in lecture-based exposition. Whenever we present a concept which is commonly mis-applied or misunderstood, we devote a few overheads to explicit warnings about the common errors. For example, a common mistake students make is to assume wrongly that the concrete state is determined by the abstract state (this arises because students learn that the abstract state is determined by the concrete state, and in many simple examples the correspondance is one-to-one). Therefore, when we define a binary search tree, we show two examples with the same combination of keys and values (that is, with the same abstract state as symbol table), but in different arrangement.

However, exposition is not in itself enough to prevent misconceptions: just as a lecture on insertion from a linked list leaves many students unsure about how to adjust the pointers, so too students will not engage with lecture discussion of the fact that the first and last elements in a list often need special case code. Thus we feel that it's essential for students to have active experience with incorrect examples, both of code and of reasoning.

Closed laboratories are usually used for students to write code; however, code reading and debugging is just as suitable for this format of teaching. We offer a range of activities where students need to deal with incorrect code for implementing some data structure. All these activities are presented by giving students a complete program, which includes the data structure code and also a test-harness which allows the data structure to be taken through a series of operations described in a textfile. The code is ready to run and reasonably documented. We try to get students to think about the topic, and read the code through, before the lab class meets. We assign "pre-work" tasks such as drawing a concrete state of the data structure after some operations (which are done correctly), identifying the variables used for a particular purpose, tracing a call of the correct part of the code, or analysing the running time of some method.

The lab itself should force students to come to terms with the error in the code. One could imagine presenting code where the student had to decide whether or not it was correct; however we do not advocate this. Instead we carefully inform students that there are errors, and ask students to find the errors, and fix them. We have adopted a character called "Fred Foolish", and we usually allocate the blame for wrong code to him; this leaves students in no doubt that the task requires identification of errors, rather than other activities such as coding or analysis.

Some misconceptions lead naturally to code that fails in almost all cases (for example, forgetting to code the base case in a recursion), but this is so obvious that students deal with it by themselves, and we don't suggest that counter-examples of this sort are worth class time. Thus all our counter-example code works correctly with many inputs. A choice faces the designer of such a lab activity: how much guidance to offer about the location of the error.

Some errors can be detected in cases that students should have learnt to test. For example, many errors with linked lists involve missing special case code to deal with boundary inputs such as an empty list, or operating on the first or last entry. For this type of counter-example, we suggest leaving students to test the code, find inputs on which it fails, trace the

code on those inputs, and thus identify precisely the location of the error and the fix that is needed.

One the other hand, some errors are hard to find. For example, in Section 5 we show code that fails only when the node being removed has an in-order successor with a right child and no left child! For this class, we suggest providing students with explicit information about an input that will show the error.

We also provide counter-examples where the misconception is expressed in reasoning rather than in code. Here we have found that a good way to structure the activity is as report-writing. Sometimes we ask students to imagine that they are consultants to an organisation where an employee has made the mistake, and the consultant needs to explain to management why this won't work. To avoid establishing stereotypes, we can vary this with counterexamples which are presented as the work of an incompetent consultant, and the student must imagine that they are an employee who needs to explain why the consultant is wrong. For other counterexamples, we use the "frame story" that the invalid argument is made by a friend, and you need to help him or her see why they made this mistake.

3 Case Study: Missing treatment of special cases

We have already mentioned the common mistake with linked lists where the code is written for the "usual case" but is missing the handling of boundary situations. We give a counter-example program that forms a class with the following header

```
/*
 * Code for sequence with cursor-based access.
 * A singly-linked list of CSNode objects
 * (without sentinels or circular links).
 * The instance variable myHead refers to
 * the CSNode representing the first element
 * of the sequence;
 * the instance variable current refers to
 * the node where the cursor is positioned
 * (current==null means that the cursor
 * is past the last element).
 */
public class CursorSeqImpl implements CursorSeq
```

We give students a complete implementation of this, including methods that print the current state, inserts an element after the cursor, etc. In Appendix A is the incorrect code we offer students, for removing an element. This fails spectacularly when one tries to remove the first element; its easy for students to find the error and correct it.

4 Case Study: Scalability confused with cost

Many students seem to believe that the algorithm with slowest growing cost will actually be the fastest in any given situation. That is, they have missed the difference between the rate of growth and the value itself.

To demonstrate this, we can ask students to respond to a proposal advocating use of a very complex but asymptotically fast algorithm in a situation where growth of the collection is not likely, because it consists only of those items which hash to a single bucket.

Here is the task

Imagine that you work for a company that sells equipment. The company web site

allows customers to look up the price of the many pieces of equipment available. The web site keeps cache copies of recently retrieved data, for the purpose of speeding up subsequent requests for this data. The cache has been coded as hash table with linear chaining. One day, your manager calls you in, to say that Fredrick Foolish Consultants have been asked to investigate the performance of the web site. The consultants' report points out that the cache includes several methods which search sequentially through a linear list of elements; they indicate that fair more efficient algorithms are known, such as red-black trees, which can speed up the cost of this type of search from linear to logarithmic. The Consultants propose that the company should employ them for a fee of \$1000 per day, to rewrite the cache management so that each linear list is replaced with a red-black tree, within the hash table.

Write a response for your manager, explaining why a red-black tree would not be a better way to structure the chain within each bucket of the table. Since your manager has not had the benefits of a Computer Science education, your response should not use complicated jargon such as "asymptotic cost" without explanation.

5 Case Study: Un-even trees

A much more complex misconception about tree structures arises from the fact that many examples in books and lectures are of small trees (to keep the diagrams manageable, perhaps). These trees are rather even in shape, close to complete trees. Students seem to assume that the natural shape of a binary tree is for each node to have two children or none. They are aware that this shape is not inevitable, and that a long linear chain is one possible shape (this is always presented as a counterexample to the elementary mistake of treating the logarithmic average cost of operations as a worst case). However, they don't consider the possibility of linear chains occurring within a bushy tree, nor do they realise that part of the structure which is linear can be a zig-zag.

To bring this home to students, and to give them real practice with tracing and debugging substantial code, we give them a binary search tree implementation whose header is

```
/*
 * Implementation of symbol table storing
 * String values, each with int key
 * using binary search tree arrangement
 * of nodes.
 * Instance variable myRoot is the root node.
 *
 * ISBSTNode is the class for each node,
 * with fields key, value, left and right.
 */
public class ISBST implements ISSymTab
```

Again we provide a complete implementation, including methods to display the contents both abstractly (as key-value pairs) and concretely (showing the tree structure), and recursive methods to lookup a given key, and to insert a key-value pair. The code contains a complicated section, involving recursion and private methods, for removing an element. The code for this aspect is in Appendix B. This code is

wrong. The error in this code is in `removeMin`, and in particular the assumption that if the element with minimum key has no left child, then it is a leaf and so can be removed without leaving descendants needing to be reconnected with the tree. We confront students with the error by asking them to run the code by inserting in turn elements with key: 6, 18, 30, 3, 12, 1, 20, 5, 24, 4, 15, 36, 22; then they are asked to remove the elements whose keys are 36, 12, 3, and 18. The first three deletions work correctly, but the removal of 18 causes the tree to lose two other entries (22 and 24). This seems to provide enough direction that the students can find what is wrong and fix it.

6 Case Study: Class invariant for BST

Our final counter-example deals with a much more specific misconception. Many students assume that the binary search tree is defined based on the relationship of each element to its children, rather than (as is actually the definition) on the relationship of each element to its descendants. This is often not picked up in the examples presented in class.

To address this we can ask students directly

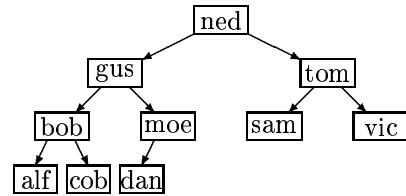
Imagine that you and your group are coding an alternate implementation for the ISBST class. You all decide that you will have a private method which will be called at the start and finish of each other call, in order to check the class invariant (this is a good approach to careful coding in “design-by-contract”). Your friend Fred Foolish is also in your group, and he offers to write the method to check the class invariant. Here is what Fred suggests as pseudocode for his recursive method

```
to check the class invariant,
call CI(myRoot)
```

```
CI(n) checks if the class invariant is true
for the subtree rooted at node n, by:
if n has a left child
    check that n.left.key < n.key
        if not, return false immediately
    check that CI(n.left)
        if not, return false immediately
if n has a right child
    check that n.key < n.right.key
        if not, return false immediately
    check that CI(n.right)
        if not, return false immediately
return true
```

Explain to Fred why his code is not the right way to check the class invariant for a binary search tree.

We hope students will spot the difference between Fred’s definition and the correct one. The difference between the definitions arises in examples such as the following, where the keys are Strings in alphabetic order. Here each element is correctly positioned compared to its children, but still this is not a BST because “dan” is found to the right of “gus”.



7 Conclusions

One contribution of this paper has been a general analysis of different ways to use counter-examples in teaching, emphasising their use in closed laboratories. We have also given several specific examples, which address some common misconceptions among students in the data structures subject. The same techniques can be used in many different subjects; our experience has covered second semester programming [5] and computer organisation [4].

8 References

- [0] ACM and IEEE, *Computing Curriculum 2001 Computer Science*, final version available at <http://www.acm.org/sigcse/cc2001/>
- [1] M. Barg, K. Crawford, A. Fekete, T. Greening, O. Hollands, J. Kay and J. Kingston “Problem-Based Learning for Foundation Computer Science courses” *Computer Science Education*, vol 10 no 2, pp 109-128, 2000.
- [2] M. Ben-Ari “Constructivism in Computer Science Education” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 257-261, 1998.
- [3] D. Dicheva, J. Close “Mental Models of Recursion” *Journal of Educational Computing Research* vol 14, no 1, pp 1-23, 1996.
- [4] A. Fekete “Enhancing Generic Skills in the Computer Organization Course” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 273-277, 1995.
- [5] A. Fekete, T. Greening, and J. Kingston, “Conveying Technical Content in a Curriculum Using Problem Based Learning” in *Proc Australasian Conference on Computer Science Education* pp 198-202, 1998.
- [6] A. Fekete, J. Kay, J. Kingston and K. Wimalaratne “Supporting Reflection in Introductory Computer Science” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 144-148, 2000.
- [7] N. Gelfand, M. Goodrich and R. Tamassia “Teaching Data Structure Design Patterns” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 331-335, 1998.
- [8] C. George “EROSI - Visualising recursion and discovering new errors” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 305-309, 2000.
- [9] D. Ginat, E. Shifroni “Teaching recursion in a procedural environment: how much should we emphasise the computing model?” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 127-131, 1999.
- [10] M. Goodrich, R. Tamassia *Data Structures and Algorithms in Java* 2nd ed, J. Wiley, 2001
- [11] S. Holland, R. Griffiths, M. Woodman “Avoiding Object Misconceptions” in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 131-134, 1997.

- [12] D. Knuth *The Art of Computer Programming 1: Fundamental Algorithms*, Addison-Wesley, 1968.
- [13] D. Nguyen, "Design Patterns for Data Structures" in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 336-340, 1998.
- [14] B. Preiss, "Design Patterns for the Data Structures and Algorithms Course" in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 95-99, 1999.
- [15] B. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, J. Wiley, 2000.
- [16] R. Rasala "A Model C++ Tree Iterator Class for Binary Search Trees" in *Proc ACM SIGCSE Symposium on Computer Science Education*, pp 72-76, 1997.
- [17] M. Weiss *Data Structures and Problem Solving Using Java*, 2nd ed, Addison-Wesley 2002.

A Code for Section 3

Here is the code which incorrectly removes an element in a singly-linked list, based on the misconception that does not deal with boundary cases.

```
/*
 * remove and return the element where the cursor is positioned;
 * leaves the cursor on the following element;
 * throws InvalidOp if the cursor is past the last element.
 */
public Object remove() throws InvalidOp
{
    if (current==null)
    {
        throw new InvalidOp();
    }
    // get here only when there is a Node to remove
    CSNode trailer; // used to find the node before the current one
    Object old; // the element being removed
    for (trailer = myHead;
         trailer!=null;
         trailer = trailer.next)
    {
        if (trailer.next==current)
        {
            break;
        }
    }
    // adjust next field in trailer node and adjust current
    old = current.data;
    trailer.next = current.next;
    current = current.next;
    return old;
}
```

B Code for Section 5

Here is the incorrect code for removing an element in a binary search tree, based on the misconception that nodes have two children or none.

```
/*
 * remove the element with given key.
 * if the key is not present, do nothing.
 * WARNING: This routine doesn't work correctly!
 */
public void delete(int key)
{
    myRoot = delete(key, myRoot);
}

/*
 * private recursive method to remove
 * the Node containing a given target key
 * in the subtree whose root is current;
 * this returns the Node which is the root of
 * the modified subtree.
 * In many cases the returned root will be the same as the
 * parameter current;
 * however when the target is found in the current node,
 * the subtree is reorganised and another node
 * is returned as the new root.
 * It maintains the BST property.
 */
private ISBSTNode delete(int target, ISBSTNode current)
{
    if (current == null)
    {
        return null;
    }

    // get here only when current != null
    if (target == current.key)
    {
```

```

    // we want to remove the current node
    // re-arranging the rest of the subtree
    // to keep the BST property depends on
    // what children the node has
    if ((current.left == null) && (current.right == null))
    {
        // current is a leaf
        return null;
    }
    if (current.right == null)
    {
        // current has only a left child
        // so remaining nodes are just subtree rooted at that child
        return current.left;
    }
    if (current.left == null)
    {
        // current has only a right child
        // so remaining nodes are just subtree rooted at that child
        return current.right;
    }
    // get here when current has two children
    // reform the subtree by moving the node with lowest key
    // from the right subtree
    // and make it take the place of current
    // its left child is current's left child
    // its right child is the result of removing it from the subtree
    // that was to the right of current
    ISBSTNode minNodeOnRight = findMin(current.right);
    ISBSTNode newRightChild = removeMin(current.right);
    ISBSTNode newLeftChild = current.left;
    minNodeOnRight.left = newLeftChild;
    minNodeOnRight.right = newRightChild;
    return minNodeOnRight;
}
else if (target < current.key)
{
    // remove from left subtree
    // attach modified subtree as left child of current
    current.left = delete(target, current.left);
    return current;
}
else // target > current.key
{
    // remove from right subtree
    // attach modified subtree as right child of current
    current.right = delete(target, current.right);
    return current;
}
}

/*
 * private recursive routine to find the node with minimum key
 * in the subtree whose root is current
 * If current is null, return null.
 */
private ISBSTNode findMin(ISBSTNode current)
{
    if (current == null)
    {
        return null;
    }

    // get here only when current != null
    if (current.left != null)
    {
        // current has left child, so find min descendant of that
        return findMin(current.left);
    }
    else // current has no left child, so min key is in current itself
    {
        return current;
    }
}

```

```

    }
}

/*
 * private recursive routine to remove the node with minimum key
 * in the subtree whose root is current.
 * this returns the node which is the root of
 * the modified subtree.
 * If current is null, return null.
 */
private ISBSTNode removeMin(ISBSTNode current)
{
    if (current == null)
    {
        return null;
    }

    // get here only when current != null
    if (current.left != null)
    {
        // current has left child, so remove min descendant of that
        // reconnect modified left subtree to left of current
        // return current node as root of modified tree
        current.left = removeMin(current.left);
        return current;
    }
    else // current has no left child, so min key is in current itself
    {
        // remove current, so return null as modified subtree
        return null;
    }
}
}

```