Pointerlab academic integrity violation

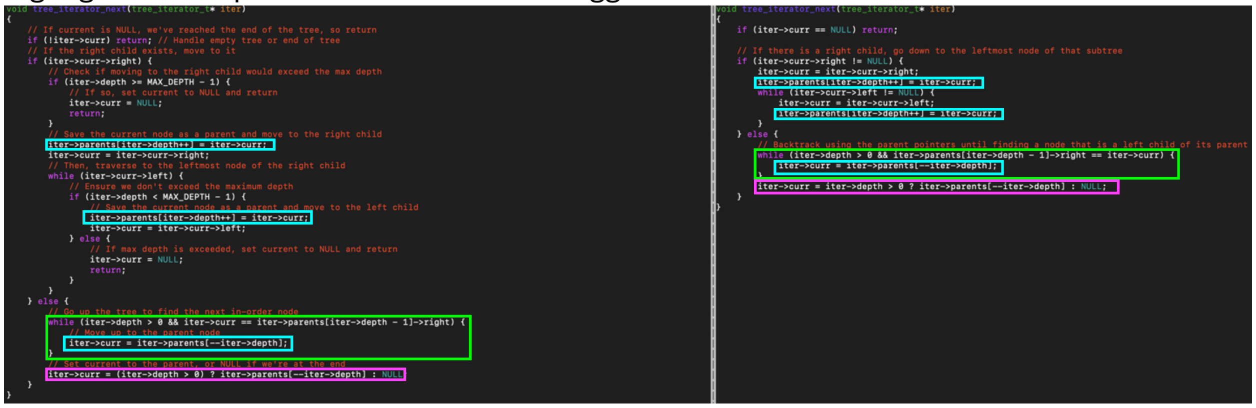
Through careful manual inspection, we have identified that the student's code matches what ChatGPT outputs. ChatGPT is an Artificial Intelligence agent that can generate code, essays, explanations, etc. based on user prompts. We have found that while it can generate different variants/versions of code, the versions are mostly similar in the code structure and style. We look for these similarities in the student's code and code generated by ChatGPT, akin to how we look for similarities between the student's code and online code or code from other students. That is, the ChatGPT code is yet another source of external code that the students may be using as the basis for their code, which is not allowed in this course.

Note that there are sometimes cosmetic differences between the student code and external code due to students modifying the code, and in the case of ChatGPT, it's possible ChatGPT or the student may be responsible for these differences. But just like how we identify violations with other online code sources, we focus on the core code structure and style similarities to demonstrate that an external resource was referenced.

Our course academic integrity policy explicitly forbids the use of ChatGPT or any external resource for anything directly related to the assignments. Since students are not allowed to get hints from any of these sources or even reference any external source, there should not be similarities between student code and these external resources. Whether the student directly utilized an Artificial Intelligence agent or indirectly referenced the code via a website, online forum, code from another student, or any other code that is based on an Artificial Intelligence agent, all of these would be considered violations. Thus, the purpose of this document is to demonstrate the code similarity, which indicates the code was not independently and individually developed by the student.

Evidence of code similarity:

Here, we present the similar code from the student (left) and ChatGPT (right). We then highlight the unique characteristics that suggest that the ChatGPT code was referenced.



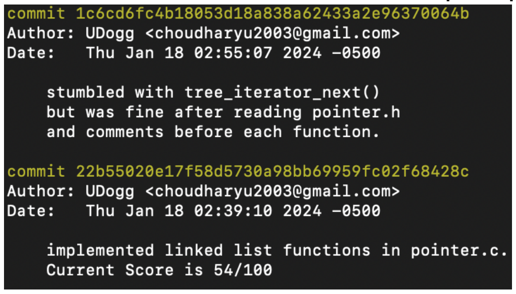
```c
void tree_iterator_next(tree_iterator_t* iter)
{
    // If current is NULL, we've reached the end of the tree, so return
    if (!iter->curr) return; // Handle empty tree or end of tree
    // If the right child exists, move to it
    if (iter->curr->right) {
        // Check if moving to the right child would exceed the max depth
        if (iter->depth >= MAX_DEPTH - 1) {
            // If so, set current to NULL and return
            iter->curr = NULL;
            return;
        }
        // Save the current node as a parent and move to the right child
        iter->parents[iter->depth++] = iter->curr;
        iter->curr = iter->curr->right;
        // Then, traverse to the leftmost node of the right child
        while (iter->curr->left) {
            // Ensure we don't exceed the maximum depth
            if (iter->depth < MAX_DEPTH - 1) {
                // Save the current node as a parent and move to the left child
                iter->parents[iter->depth++] = iter->curr;
                iter->curr = iter->curr->left;
            } else {
                // If max depth is exceeded, set current to NULL and return
                iter->curr = NULL;
                return;
            }
        }
    } else {
        // Go up the tree to find the next in-order node
        while (iter->depth > 0 && iter->curr == iter->parents[iter->depth - 1]->right) {
            // Move up to the parent node
            iter->curr = iter->parents[--iter->depth];
        }
        // Set current to the parent, or NULL if we're at the end
        iter->curr = (iter->depth > 0) ? iter->parents[--iter->depth] : NULL
    }
}
```

```c
void tree_iterator_next(tree_iterator_t* iter)
{
    if (iter->curr == NULL) return;

    // If there is a right child, go down to the leftmost node of that subtree
    if (iter->curr->right != NULL) {
        iter->curr = iter->curr->right;
        iter->parents[iter->depth++] = iter->curr;
        while (iter->curr->left != NULL) {
            iter->curr = iter->curr->left;
            iter->parents[iter->depth++] = iter->curr;
        }
    } else {
        // Backtrack using the parent pointers until finding a node that is a left child of its parent
        while (iter->depth > 0 && iter->parents[iter->depth - 1]->right == iter->curr) {
            iter->curr = iter->parents[--iter->depth];
        }
        iter->curr = iter->depth > 0 ? iter->parents[--iter->depth] : NULL;
    }
}
```

1. What stands out in this code are the lines with iter->parents[iter->depth++] and iter->parents[--iter->depth] (cyan boxes). This uses a post-increment and pre-decrement operators, which are an advanced C style that most students are

unfamiliar with. Most experienced programmers avoid this style since it can be confusing and makes the code harder to read. For this particular piece of code, it seems that ChatGPT tends to generate code with this style, whereas a typical human programmer would adopt different styles.

2. Another part that stands out is the identical while loop conditions (green box). It's a complex loop condition that happens to match nearly exactly with only the ordering of the second condition equality switched, which doesn't impact the code. The contents of the while loop are also equivalent.
3. The code in the magenta box is also equivalent, including the use of the ternary notation with the ? and : symbols. The ternary notation is yet another uncommon code style that students aren't expected to know or use. It is more common than the post/pre increment/decrement use cases in the cyan boxes, but it's an uncommon use case that stands out as unusually similar.
4. Lastly, note that the student implemented all of the tree iterator code (including this function, which is the most complex part of the assignment) in just 16 minutes.

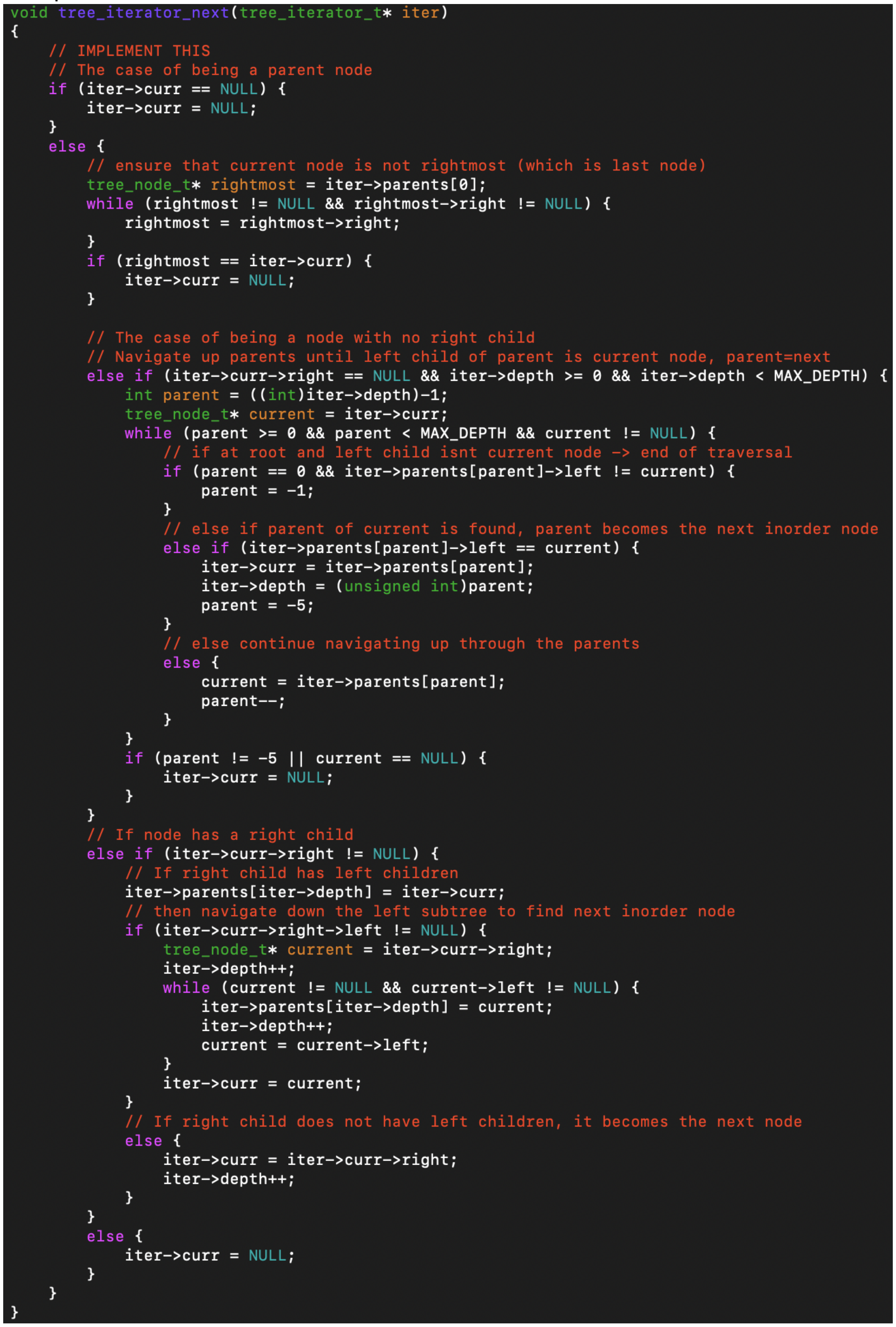


```
commit 1c6cd6fc4b18053d18a838a62433a2e96370064b
Author: UDogg <choudharyu2003@gmail.com>
Date:   Thu Jan 18 02:55:07 2024 -0500

    stumbled with tree_iterator_next()
    but was fine after reading pointer.h
    and comments before each function.

commit 22b55020e17f58d5730a98bb69959fc02f68428c
Author: UDogg <choudharyu2003@gmail.com>
Date:   Thu Jan 18 02:39:10 2024 -0500

    implemented linked list functions in pointer.c.
    Current Score is 54/100
```

This is just unrealistically fast. Implementing, testing, and debugging this part of the code can take some students days, and the fastest students would take many hours. For reference, one of my TAs took about 5 hours on this part. As described above, the code was implemented in a complex style that makes it difficult to implement and debug, and it uses features most likely unknown to students. This makes it even less believable that this could be done in 16 minutes without any external assistance. Given that the code also matches what ChatGPT outputs, I'm confident the code was based on some external source, whether it be ChatGPT directly or some other external source that was indirectly based on ChatGPT.

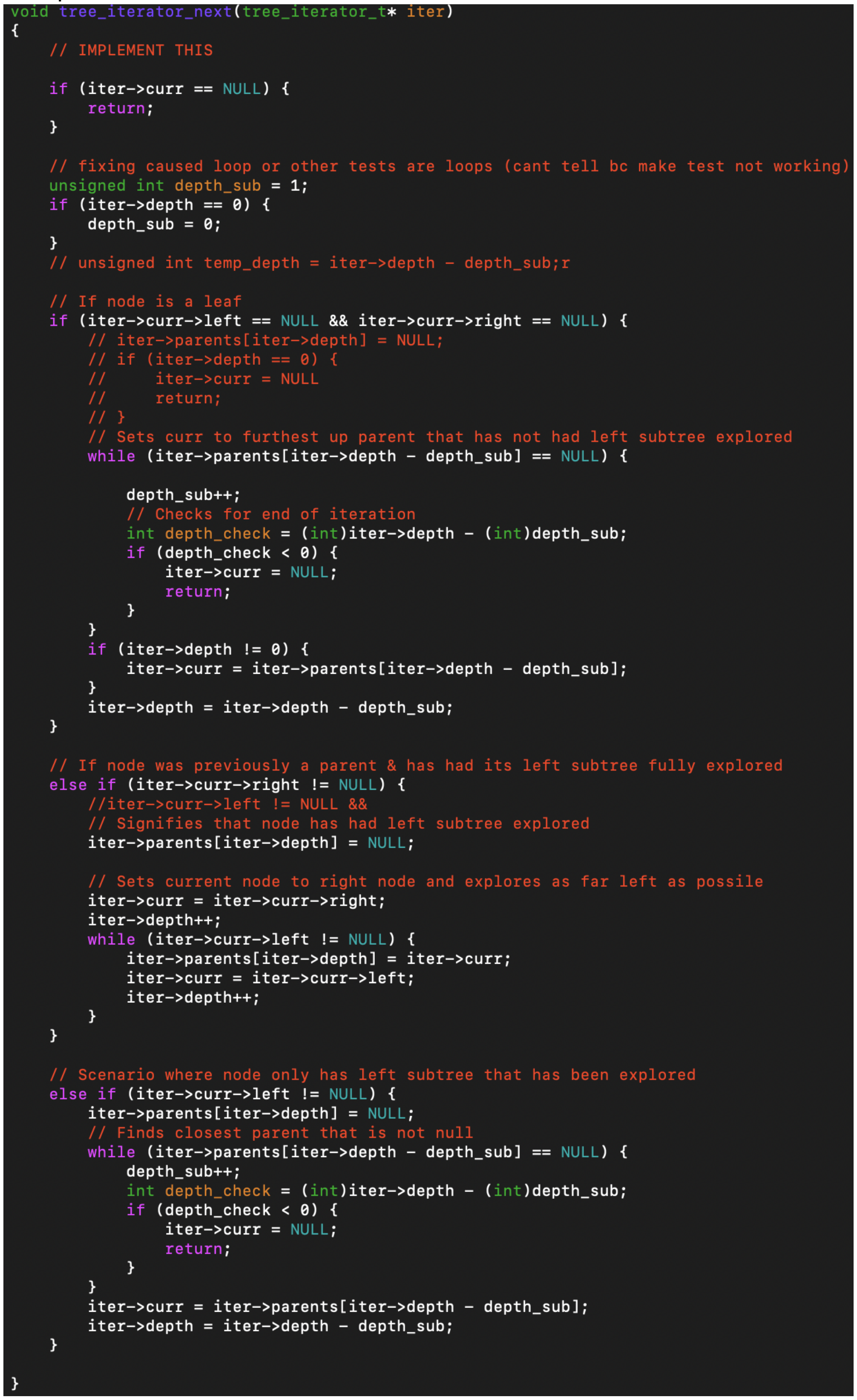
There are many ways to write the code:
Here, we present a few other ways of writing the code to demonstrate that there are many correct ways of doing the assignment. We show 3 samples from student code, but there are many other ways of writing the code. If the academic integrity committee would like additional examples, please let me know.

Sample 1:



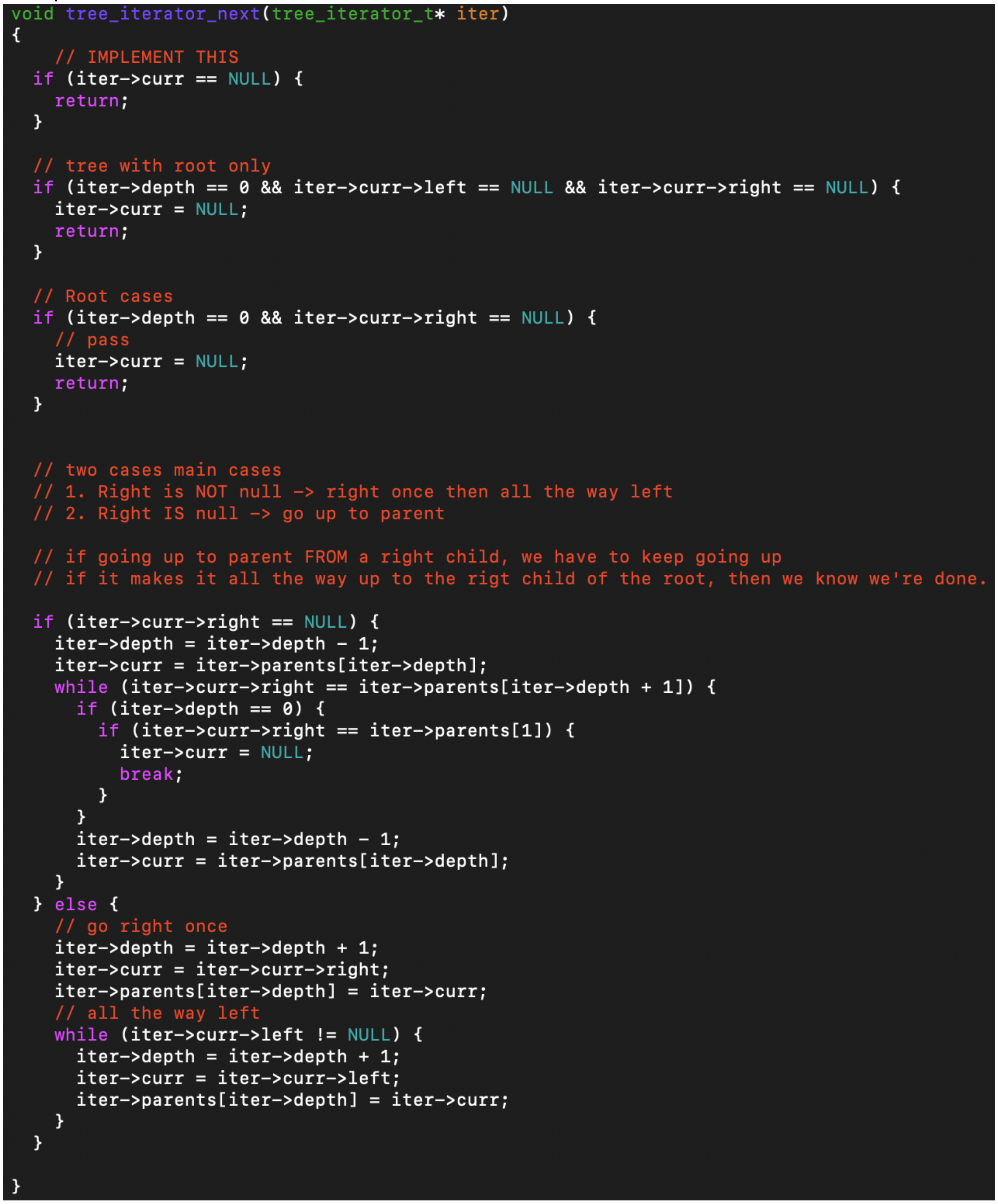
```c
void tree_iterator_next(tree_iterator_t* iter)
{
    // IMPLEMENT THIS
    // The case of being a parent node
    if (iter->curr == NULL) {
        iter->curr = NULL;
    }
    else {
        // ensure that current node is not rightmost (which is last node)
        tree_node_t* rightmost = iter->parents[0];
        while (rightmost != NULL && rightmost->right != NULL) {
            rightmost = rightmost->right;
        }
        if (rightmost == iter->curr) {
            iter->curr = NULL;
        }

        // The case of being a node with no right child
        // Navigate up parents until left child of parent is current node, parent=next
        else if (iter->curr->right == NULL && iter->depth >= 0 && iter->depth < MAX_DEPTH) {
            int parent = ((int)iter->depth)-1;
            tree_node_t* current = iter->curr;
            while (parent >= 0 && parent < MAX_DEPTH && current != NULL) {
                // if at root and left child isnt current node -> end of traversal
                if (parent == 0 && iter->parents[parent]->left != current) {
                    parent = -1;
                }
                // else if parent of current is found, parent becomes the next inorder node
                else if (iter->parents[parent]->left == current) {
                    iter->curr = iter->parents[parent];
                    iter->depth = (unsigned int)parent;
                    parent = -5;
                }
                // else continue navigating up through the parents
                else {
                    current = iter->parents[parent];
                    parent--;
                }
            }
            if (parent != -5 || current == NULL) {
                iter->curr = NULL;
            }
        }
        // If node has a right child
        else if (iter->curr->right != NULL) {
            // If right child has left children
            iter->parents[iter->depth] = iter->curr;
            // then navigate down the left subtree to find next inorder node
            if (iter->curr->right->left != NULL) {
                tree_node_t* current = iter->curr->right;
                iter->depth++;
                while (current != NULL && current->left != NULL) {
                    iter->parents[iter->depth] = current;
                    iter->depth++;
                    current = current->left;
                }
                iter->curr = current;
            }
            // If right child does not have left children, it becomes the next node
            else {
                iter->curr = iter->curr->right;
                iter->depth++;
            }
        }
        else {
            iter->curr = NULL;
        }
    }
}
```

Sample 2:



```
void tree_iterator_next(tree_iterator_t* iter)
{
    // IMPLEMENT THIS

    if (iter->curr == NULL) {
        return;
    }

    // fixing caused loop or other tests are loops (cant tell bc make test not working)
    unsigned int depth_sub = 1;
    if (iter->depth == 0) {
        depth_sub = 0;
    }
    // unsigned int temp_depth = iter->depth - depth_sub;r

    // If node is a leaf
    if (iter->curr->left == NULL && iter->curr->right == NULL) {
        // iter->parents[iter->depth] = NULL;
        // if (iter->depth == 0) {
        //     iter->curr = NULL
        //     return;
        // }
        // Sets curr to furthest up parent that has not had left subtree explored
        while (iter->parents[iter->depth - depth_sub] == NULL) {

            depth_sub++;
            // Checks for end of iteration
            int depth_check = (int)iter->depth - (int)depth_sub;
            if (depth_check < 0) {
                iter->curr = NULL;
                return;
            }
        }
        if (iter->depth != 0) {
            iter->curr = iter->parents[iter->depth - depth_sub];
        }
        iter->depth = iter->depth - depth_sub;
    }

    // If node was previously a parent & has had its left subtree fully explored
    else if (iter->curr->right != NULL) {
        //iter->curr->left != NULL &&
        // Signifies that node has had left subtree explored
        iter->parents[iter->depth] = NULL;

        // Sets current node to right node and explores as far left as possile
        iter->curr = iter->curr->right;
        iter->depth++;
        while (iter->curr->left != NULL) {
            iter->parents[iter->depth] = iter->curr;
            iter->curr = iter->curr->left;
            iter->depth++;
        }
    }

    // Scenario where node only has left subtree that has been explored
    else if (iter->curr->left != NULL) {
        iter->parents[iter->depth] = NULL;
        // Finds closest parent that is not null
        while (iter->parents[iter->depth - depth_sub] == NULL) {
            depth_sub++;
            int depth_check = (int)iter->depth - (int)depth_sub;
            if (depth_check < 0) {
                iter->curr = NULL;
                return;
            }
        }
        iter->curr = iter->parents[iter->depth - depth_sub];
        iter->depth = iter->depth - depth_sub;
    }

}
```

Sample 3:



```c
void tree_iterator_next(tree_iterator_t* iter)
{
    // IMPLEMENT THIS
  if (iter->curr == NULL) {
    return;
  }

  // tree with root only
  if (iter->depth == 0 && iter->curr->left == NULL && iter->curr->right == NULL) {
    iter->curr = NULL;
    return;
  }

  // Root cases
  if (iter->depth == 0 && iter->curr->right == NULL) {
    // pass
    iter->curr = NULL;
    return;
  }


  // two cases main cases
  // 1. Right is NOT null -> right once then all the way left
  // 2. Right IS null -> go up to parent

  // if going up to parent FROM a right child, we have to keep going up
  // if it makes it all the way up to the rigt child of the root, then we know we're done.

  if (iter->curr->right == NULL) {
    iter->depth = iter->depth - 1;
    iter->curr = iter->parents[iter->depth];
    while (iter->curr->right == iter->parents[iter->depth + 1]) {
      if (iter->depth == 0) {
        if (iter->curr->right == iter->parents[1]) {
          iter->curr = NULL;
          break;
        }
      }
      iter->depth = iter->depth - 1;
      iter->curr = iter->parents[iter->depth];
    }
  } else {
    // go right once
    iter->depth = iter->depth + 1;
    iter->curr = iter->curr->right;
    iter->parents[iter->depth] = iter->curr;
    // all the way left
    while (iter->curr->left != NULL) {
      iter->depth = iter->depth + 1;
      iter->curr = iter->curr->left;
      iter->parents[iter->depth] = iter->curr;
    }
  }

}
```