UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 5

Node chains — Solutions

Date Due: March 2, 2018, 10pm **Total Marks: 50**

Question 0 (5 points):

Purpose: To force the use of Version Control in Assignment 5

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 5. This is a tool that you need to be required to use, even when you don't need it, so that when you do need it, you are already familiar with it. Do the following steps.

1. Create a new PyCharm project for Assignment 5.
2. Use `Enable Version Control Integration...` to initialize Git for your project.
3. Download the Python and text files provided for you with the Assignment, and add them to your project.
4. Before you do any coding or start any other questions, make an initial commit.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open the terminal in your Assignment 5 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm, locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.

   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A3 folder, and run `git --no-pager log` there. If you did all your work in this folder, git will be able to see it even if you did your work on Windows. Git's information is out of sight, but in your folder.

   Note: If you are working at home on Windows, Google for how to make git available on your command-line window. You basically have to tell the command-line app where the git app is.

You may need to work in the lab for this; Git is installed there. Not having Git installed is not really an excuse. It's like driving a car without wearing a seatbelt. It's not an excuse to say "My car doesn't have a seatbelt."

What to Hand In

After completing and submitting your work for Questions 1-3, open a command-line window in your Assignment 5 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/-paste this into a text file named `a5-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the way to use git, but it is the way students work on assignments.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: The log file shows that you used Git as part of your work for Assignment 5. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per question for a total of at least 6 commits. And frankly, if you only have 6 commits, you're pretending.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 3 marks for good commit messages; 2 marks for anything else.
- Students were told at least 6 commits (2 per question).
- Give 2 marks for 6 commits or more, and 1 marks for anything less than 6.

Question 1 (5 points):

Purpose: To practice debugging a function that works with node chains created using the Node ADT.

Degree of Difficulty: Easy

On Moodle, you will find a *starter file* called `a5q1.py`. It has a broken implementation of the function `to_string()`, which is a function used by the rest of the assignment.

You will also find a test script named `a5q1_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

Debug and fix the function `to_string()`. The error in the function is pretty typical of novice errors with this kind of programming task.

The interface for the function is:

```
def to_string(node_chain):
    """
    Purpose:
        Create a string representation of the node chain.  E.g.,
        [ 1 | *-]-->[ 2 | *-]-->[ 3 | / ]
    Pre-conditions:
        :param node_chain:  A node-chain, possibly empty
    Post_conditions:
        None
    Return: A string representation of the nodes.
    """
```

Note carefully that the function does not do any console output. It should return a string that represents the node chain.

Here's how it might be used:

```
empty_chain = None
chain = node.create(1, node.create(2, node.create(3)))

print('empty_chain --->', to_string(empty_chain))
print('chain --------->', to_string(chain))
```

Here's what the above code is supposed to do when the function is working:

```
empty_chain ---> EMPTY
chain ---------> [ 1 | *-]-->[ 2 | *-]-->[ 3 | / ]
```

Notice that the string makes use of the characters `'[ | *-]-->'` to reflect the chain of references. The function also uses the character `'/'` to abbreviate the value `None` that indicates the end of a chain. Note especially that the empty chain is represented by the string `'EMPTY'`.

What to Hand In

A file named `a5q1.py` with the corrected definition of the function. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: The function `to_string()` works correctly

Solution: Here's a corrected implementation of `to_string()`.

```
def to_string(node_chain):
    """
    Purpose:
        Create a string representation of the node chain.  E.g.,
        [ 1 | *-]-->[ 2 | *-]-->[ 3 | / ]
    Pre-conditions:
        :param node_chain:  A node-chain, possibly empty
    Post_conditions:
        None
    Return: A string representation of the nodes.
    """
    # special case: empty node chain
    if node_chain is None:
        result = 'EMPTY'
    else:
        # walk along the chain
        walker = node_chain
        value = node.get_data(walker)
        # print the data
        result = '[ ' + str(value) + ' |'
        while node.get_next(walker) is not None:
            walker = node.get_next(walker)
            value = node.get_data(walker)
            # represent the next with an arrow-like figure
            result += ' *-]-->[ '+str(value)+' |'

        # at the end of the chain, use '/'
        result += ' / ]'

    return result
```

An error was introduced in the while-loop condition. A simple fix is to check `get_next(walker)` instead of `walker` directly.

There are other ways to fix the function by making bigger changes.

Marking guidelines:

- Give full marks if the function works. The simplest fix seems obvious, but I am sure some will make bigger changes.

Question 2 (20 points):

Purpose: To practice working with node chains created using the Node ADT. These functions do not make any changes to the node chains.

Degree of Difficulty: Easy.

In this question you'll write three functions for node-chains that are a little more challenging. On Moodle, you can find a *starter file* called `a5q2.py`, with all the functions and doc-strings in place, and your job is to write the bodies of the functions. You will also find a test script named `a5q2_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

(a) (5 points) Implement the function `count_chain()`. The interface for the function is:

```
def count_chain(node_chain):
    """
    Purpose:
        Counts the number of nodes in the node chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
    Return:
        :return: The number of nodes in the node chain.
    """
```

Note carefully that the function is not to do any console output.
A demonstration of the application of the function is as follows:

```
empty_chain = None
chain = node.create(1, node.create(2, node.create(3)))

print('empty chain has', count_chain(empty_chain), 'elements')
print('chain has', count_chain(chain), 'elements')
```

The output from the demonstration is as follows:

```
empty chain has 0 elements
chain has 3 elements
```

(b) (5 points) Implement the function `copy_chain()`. The interface for the function is:

```
def copy_chain(node_chain):
    """
    Purpose:
        Make a new node chain with the same values as in node_chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly None
    Return:
        :return: A copy of node chain, with new nodes, but the same data.
    """
```

The function `copy_chain()` should create a sequence of new nodes, with the same data values as the given chain.

A demonstration of the application of the function is as follows:

```
chain = node.create(1, node.create(2, node.create(3)))
chain4 = copy_chain(chain)

print("chain and chain4 are different chains?", chain is not chain4)
print("chain and chain4 have the same values?", chain == chain4)
```

Remember that `is` checks for equality of reference.
The output from the demonstration is as follows:

```
chain and chain4 are different chains? True
chain and chain4 have the same values? True
```

The output from the first print statement indicates that the nodes are different, but the second statement indicates they have the same values.

(c) (5 points) Implement the function `replace()`. The interface for the function is:

```
def replace(node_chain, target, value):
    """
    Purpose:
        Replace every occurrence of data target in node_chain with data value
        The chain's data may change, but the node structure will not.
    Pre-Conditions:
        :param node_chain: a node-chain, possibly empty
        :param target: a data value
        :param value: a data value
    Post-conditions:
        The node-chain is changed, by replacing target with value everywhere.
    Return:
        :return: None
    """
```

This function affects the data values stored in the node-chain, but should not change any of the nodes in the chain. If the target does not appear in the node-chain, the node-chain is not modified at all.
A demonstration of the application of the function is as follows:

```
chain5 = node.create(3,
         node.create(1,
         node.create(4,
         node.create(1,
         node.create(5)))))

print('before:', to_string(chain5))
replace(chain5, 1, 8)
print('after: ', to_string(chain5))
```

The output from the demonstration is as follows:

```
before: [ 3 | *-]-->[ 1 | *-]-->[ 4 | *-]-->[ 1 | *-]-->[ 5 | / ]
after:  [ 3 | *-]-->[ 8 | *-]-->[ 4 | *-]-->[ 8 | *-]-->[ 5 | / ]
```

(d) (5 points) Before you submit your work, review it, and edit it for programming style. Make sure your variables are named well, and that you have appropriate (not excessive) internal documentation (do not change the doc-string, which we have given you).

What to Hand In

A file named `a5q2.py` with the definitions of the ~~four~~ three functions. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your function `count_chain()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to return the number of nodes in the chain correctly.
  - Works on node-chains of any length.
- 5 marks: Your function `copy_chain()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to return a sequence of new nodes containing the same data as the given node-chain.
  - Works on node-chains of any length.
- 5 marks: Your function `replace()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to replace every target value appropriately, while leaving the node-chain intact.
  - Works on node-chains of any length.
- 5 marks: Overall, you used good programming style, including:
  - Good variable names
  - Appropriate internal comments (outside of the given doc-strings)

Solution:

(a) Here's an implementation of `count_chain()` that would be acceptable.

```
def count_chain(node_chain):
    """
    Purpose:
        Counts the number of nodes in the node chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
    Return:
        :return: The number of nodes in the node chain.
    """
    if node_chain is None:
        return 0
    else:
        return 1 + count_chain(node.get_next(node_chain))
```

My implementation used recursion, but a loop with a walker is completely acceptable as well.

```
def count_chain_v2(node_chain):
    """
    Purpose:
        Counts the number of nodes in the node chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
    Return:
        :return: The number of nodes in the node chain.
    """
    walker = node_chain
    counter = 0
    while walker is not None:
        walker = node.get_next(walker)
        counter += 1
    return counter
```

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only (recursively or with a loop)
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.
- No testing or demonstration is required.

(b) Here's an implementation of `copy_chain()` that would be acceptable.

```
def copy_chain(node_chain):
    """
    Purpose:
        Make a new node chain with the same values as in node_chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly None
    Return:
        :return: A copy of node chain, with new nodes, but the same data.
    """
    # special case: empty node chain
    if node_chain is None:
        return None
    else:
        walker = node_chain

        # remember the first node
        result = node.create(node.get_data(walker))
        walker = node.get_next(walker)
        walker2 = result

        # walk along the first chain, making copies of each node
        while walker is not None:
            node.set_next(walker2, node.create(node.get_data(walker)))
            walker2 = node.get_next(walker2)
            walker = node.get_next(walker)

        # return the anchor to the copy
        return result
```

The special case of an empty chain is necessary. Otherwise, the function walks along the given node-chain in the usual manner. The only tricky part here is that the new node chain needs an anchor that doesn't move, but is returned when the copying is done.

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only.
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.
- No testing or demonstration is required.

(c) Here's an implementation of `replace()` that would be acceptable.

```
def replace(node_chain, target, value):
    """
    Purpose:
        Replace every occurrence of data target in node_chain with data value
        The chain's data may change, but the node structure will not.
    Pre-Conditions:
        :param node_chain: a node-chain, possibly empty
        :param target: a data value
        :param value: a data value
    Post-conditions:
        The node-chain is changed, by replacing target with value everywhere.
    Return:
        :return: None
    """
    walker = node_chain
    while walker is not None:
        if node.get_data(walker) == target:
            node.set_data(walker, value)
        walker = node.get_next(walker)
```

The function walks along the given node-chain in the usual manner. There is nothing tricky here.

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only.
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.
- No testing or demonstration is required.

(d) Programming style. Students have been advised on good style (see Chapter 9 in the textbook). They should be using appropriate variable names (not too long and not too short), and have appropriate levels of comment in the code (not too much and not too little). The given model solutions show an appropriate level of both.

Marking guidelines:

- Deduct 3 marks if variable names are not appropriate.
- Deduct 2 marks if there are no comments at all, or if there is excessive commenting.

Question 3 (20 points):

Purpose: To practice working with node chains created using the Node ADT. These functions change the node structure!

Degree of Difficulty: Moderate to Tricky.

In this question you'll write some functions for node-chains that are even more challenging. On Moodle, you can find a *starter file* called `a5q3.py`, with all the functions and doc-strings in place, and your job is to write the bodies of the functions. You will also find a test script named `a5q3_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

(a) (5 points) Implement the function `split_chain()`. The interface for the function is:

```
def split_chain(node_chain):
    """
    Purpose:
        Splits the given node chain in half, returning the second half.
        If the given chain has an odd length, the extra node is part of
        the second half of the chain.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Post-conditions:
        the original node chain is cut in half!
    Return:
        :return: A tuple (nc1, nc2) where nc1 and nc2 are node-chains
         each containing about half of the nodes in node-chain
    """
```

This function should not create any nodes, and should return a tuple of references to the two halves of the chain. The tricky part of this is only to remember to put a `None` at the right place, to terminate the original node-chain about half way through.

A demonstration of the application of the function is as follows:

```
chain5 = node.create(3,
         node.create(1,
         node.create(4,
         node.create(1,
         node.create(5)))))

print('chain5 Before:', to_string(chain5))

a, b = split_chain(chain5)
print('chain5 After: ', to_string(a))
print('Second half:  ', to_string(b))
```

The output from the demonstration is as follows:

```
chain5 Before: [ 3 | *-]-->[ 1 | *-]-->[ 4 | *-]-->[ 1 | *-]-->[ 5 | / ]
chain5 After:  [ 3 | *-]-->[ 1 | / ]
Second half:   [ 4 | *-]-->[ 1 | *-]-->[ 5 | / ]
```

Notice how the second half of the list is a little bit larger.

Hint: You may use `count_chain()` from A5Q2, and integer division.

(b) (5 points) Implement the function `remove_chain()`. The interface for the function is:

```
def remove_chain(node_chain, val):
    """
    Purpose:
        Remove the first occurrence of val from node_chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
        :param val: a value to be removed
    Post-conditions:
        The first occurrence of the value is removed from the chain.
        If val does not appear, the node-chain is unmodified.
    Return:
        :return: The resulting node chain with val removed
    """
```

Note that the function should return the original node-chain if the value does not occur in the node-chain.

A demonstration of the application of the function is as follows:

```
chain6 = node.create(7, node.create(3, node.create(4, node.create(3))))

print('before:', to_string(chain6))
chain6 = remove_chain(chain6, 3)
print('after: ',to_string(chain6))
```

The output from the demonstration is as follows:

```
before: [ 7 | *-]-->[ 3 | *-]-->[ 4 | *-]-->[ 3 | / ]
after:  [ 7 | *-]-->[ 4 | *-]-->[ 3 | / ]
```

Notice that only the first of the 3s were removed from the node-chain.

Hint: there are 2 special cases: when the value is first in the node-chain, and when it is last in the node-chain.

(c) (5 points) Implement the function `insert_at()`. The interface for the function is:

```
def insert_at(node_chain, value, index):
    """
    Purpose:
        Insert the given value into the node-chain so that
        it appears at the the given index.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
        :param value: a value to be inserted
        :param index: the index where the new value should appear
        Assumption:  0 <= index <= n
                     where n is the number of nodes in the chain
    Post-condition:
        The node-chain is modified to include a new node at the
        given index with the given value as data.
    Return
        :return: the node-chain with the new value in it
    """
```

Note carefully that the index is assumed to be in the range from 0 to n, where n is the length of the node-chain. Your function does not have to check this (but you may use an assertion here). Given an index of 0, the function puts the new value first, and given an index of n, it puts the new value last.

A demonstration of the application of the function is as follows:

```
empty_chain = None
one_node = node.create(5)
chain7 = node.create(5, node.create(7, node.create(11)))

print('Before:', to_string(empty_chain))
print('Before:', to_string(one_node))
print('Before:', to_string(chain7))

print('After:', to_string(insert_at(empty_chain, 'here', 0)))
print('After:', to_string(insert_at(one_node, 'here', 0)))
print('After:', to_string(insert_at(chain7, 'here', 1)))
print('After:', to_string(insert_at(chain7, 'again', 4)))
```

The output from the demonstration is as follows:

```
Before: EMPTY
Before: [ 5 | / ]
Before: [ 5 | *-]-->[ 7 | *-]-->[ 11 | / ]
After: [ here | / ]
After: [ here | *-]-->[ 5 | / ]
After: [ 5 | *-]-->[ here | *-]-->[ 7 | *-]-->[ 11 | / ]
After: [ 5 | *-]-->[ here | *-]-->[ 7 | *-]-->[ 11 | *-]-->[ again | / ]
```

Note that the variable `chain7` above refers to a node-chain that has had two values inserted into it.

(d) (5 points) Before you submit your work, review it, and edit it for programming style. Make sure your variables are named well, and that you have appropriate (not excessive) internal documentation (do not change the doc-string, which we have given you).

What to Hand In

A file named `a5q3.py` with the definition of your functions. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your function `split_chain()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to divide the existing node chain in two roughly equal halves.
  - Works on node-chains of any length.
- 5 marks: Your function `remove_chain()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to remove the first node containing the given value from the chain.
  - Works on node-chains of any length.
- 5 marks: Your function `insert_at()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to insert a new node with the given value so that it appears at the given index.
  - Works on node-chains of any length.
- 5 marks: Overall, you used good programming style, including:
  - Good variable names
  - Appropriate internal comments (outside of the given doc-strings)

Solution:

(a) Here's an implementation of `split_chain()` that would be acceptable.

```
def split_chain(node_chain):
    """
    Purpose:
        Splits the given node chain in half, returning the second half.
        If the given chain has an odd length, the extra node is part of
        the second half of the chain.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Post-conditions:
        the original node chain is cut in half!
    Return:
        :return: A tuple (nc1, nc2) where nc1 and nc2 are node-chains
         each containing about half of the nodes in node-chain
    """
    # calculate size
    n = a5q2.count_chain(node_chain)

    # special case: empty chain
    if n == 0:
        return None, None
    # special case: chain with one node
    if n == 1:
        return None, node_chain

    # calculate halfway
    mid = n // 2

    # walk along the chain until half-way
    walker = node_chain
    prev = None
    counter = 0
    while counter < mid:
        prev = walker
        walker = node.get_next(walker)
        counter += 1

    # terminate the first half with a well-placed None
    node.set_next(prev, None)

    # return the two new chains
    return node_chain, walker
```

The function walks along the given node-chain in the usual manner. The splitting is accomplished by placing a `None` in the last node of the original chain, and returning the reference to the node that starts the second half.

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only.
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.

- No testing or demonstration is required.

(b) Here's an implementation of `remove_chain()` that would be acceptable.

```
def remove_chain(node_chain, val):
    """
    Purpose:
        Remove the first occurrence of val from node_chain.
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
        :param val: a value to be removed
    Post-conditions:
        The first occurrence of the value is removed from the chain.
        If val does not appear, the node-chain is unmodified.
    Return:
        :return: The resulting node chain with val removed
    """
    # special case: empty node chain
    if node_chain is None:
        return None

    # special case: node to be removed is first
    if node.get_data(node_chain) == val:
        return node.get_next(node_chain)

    # walk along the chain
    walker = node.get_next(node_chain)
    prev = node_chain
    while walker is not None and node.get_data(walker) != val:
        prev = walker
        walker = node.get_next(walker)

    # found first occurrence of val?
    if walker is not None:
        node.set_next(prev, node.get_next(walker))

    return node_chain
```

The special case of an empty node-chain is handled first. After that, we check another special case: the chain starts with a node containing the value to be removed. There could be several repeated nodes with that value, so we keep advancing until we find one that isn't to be removed. This node will be the anchor of the modified chain.

After that, we step along the node chain to the end of the chain. We use the technique of two walkers, one behind the other, so that we can use `set_next()` to connect the previous node to the next node if `val` is ever seen as a data value in the chain.

If we reach the end of the node-chain before we find `val`, then we simply return the original node chain.

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only.
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.
- No testing or demonstration is required.

(c) Here's an implementation of `insert_at()` that would be acceptable.

```
def insert_at(node_chain, value, index):
    """
    Purpose:
        Insert the given value into the node-chain so that
        it appears at the the given index.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
        :param value: a value to be inserted
        :param index: the index where the new value should appear
        Assumption:  0 <= index <= n
                     where n is the number of nodes in the chain
    Post-condition:
        The node-chain is modified to include a new node at the
        given index with the given value as data.
    Return
        :return: the node-chain with the new value in it
    """

    # special case: insert at index 0
    if index == 0:
        return node.create(value, node_chain)

    # walk along the chain until the indicated index
    walker = node_chain
    counter = 0
    prev = None
    while walker is not None and counter < index:
        prev = walker
        walker = node.get_next(walker)
        counter += 1

    # If there is a node at the index
    if counter == index:
        # insert a new node
        node.set_next(prev, node.create(value, walker))
    else:
        # insert at the very end
        node.set_next(prev, node.create(value, None))

    return node_chain
```

The special case of inserting at index 0 is handled first; this should always be possible. After that, we use a pair of walkers to step along the chain until the right position is reached. The implementation assumes that the given index is in the correct range. It would be acceptable to add assertions to indicate when indices are out of range.

Marking guidelines:

- For full marks, the function must walk along the chain using the Node ADT only.
- Deduct 3 marks if the function violates the Node ADT, or if Python lists were used somehow.
- No testing or demonstration is required.

(d) Programming style. Students have been advised on good style (see Chapter 9 in the textbook). They should be using appropriate variable names (not too long and not too short), and have appropriate levels of comment in the code (not too much and not too little). The given model solutions show an appropriate level of both.

Marking guidelines:

- Deduct 3 marks if variable names are not appropriate.
- Deduct 2 marks if there are no comments at all, or if there is excessive commenting.