UNIVERSITY OF SASKATCHEWAN

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

CMPT 145
Spring-Summer 2021
Principles of Computer Science

# Assignment 5 – Solutions and Grading

## Nodes and Node Chains

**Date Due: Wednesday, June 23, 11:59pm** **Total Marks: 62**

## Question 1 (10 points):

**Learning Objectives**

**Purpose:** Students will practice the following skills:

- Debugging a function that works with node chains created using the Node ADT.
- Developing a practical tool for use in future questions.

**Degree of Difficulty:** Easy.

**References:** You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes

**Restrictions:** This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

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 you could use in ~~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 (None)
    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 = N.node(1, N.node(2, N.node(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'.

## Questions

Answer the following questions about nodes, and node-chains:

1. Suppose we create a node chain as follows:

   ```
   example = N.node(1, N.node(2, N.node(3)))
   ```

   What happens when we call `print(example)`? Explain what the displayed information is telling you, in your own words.

2. Suppose you are debugging a node-chain application, and you wanted to see the contents of a node-chain named `example`, Would you prefer to use `print(example)` or `print(to_string(example))`? Why? Assume that the `to_string()` is fixed before you use it.

3. Suppose you were writing test cases for a function `to_chain()` that creates a node chain from values stored in a list. For example:

   ```
   example2 = to_chain([1,2,3])
   ```

   You don't have the code for this function, but assume it returns a node chain very similar to the one above. How could you use `to_string()` in test cases for `to_chain()`? Remember that we want the computer to do all the checking. To answer this question, give one test case for the (missing) function `to_chain()` that shows you understand how to use `to_string()` for testing.

## What to Hand In

- A file named `a5q1.py` with the corrected definition of the function.
- A file named `a5q1_answers.txt` with the answers to the questions above.

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

## Evaluation

- 4 marks: The function `to_string()` works correctly
- 6 marks: Your answers to the questions demonstrate that you understand the value of `to_string()` in debugging and testing.

**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 (None)
    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 = walker.get_data()
        # print the data
        result = '[ {} |'.format(str(value))
        while walker.get_next() is not None:
            walker = walker.get_next()
            value = walker.get_data()
            # represent the next with an arrow-like figure
            result += ' *-]-->[ {} |'.format(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.

**Questions answered**

1. If we call `print(example)`, Python will display something like

```
<node.node object at 0x7f9d300315f8>
```

   This tells us that there is a node object in memory, and the address is given. However, it does not tell us anything informative about the node.

2. I would prefer to see what's in the chain, because it will help me understand the structure better.

3. We can sometimes use `to_string()` to see if the node chain we get back from a function has the right values and structure. For example:

```
input = [1, 2, 3]
expected = '[ 1 | *-]-->[ 2 | *-]-->[ 3 | / ]'

result = to_chain(input)
if expected != to_string(result):
    print('Error')
```

**Marking guidelines:**

- Give full marks if the function works. The simplest fix seems obvious, but I am sure some will make bigger changes.
- For the questions:
  1. Any answer talking about the location of the object, but not its structure gets full marks.
  2. The answer has to mention that knowing the structure of a node chain is valuable for debugging and testing.
  3. Comparing the output of `to_string()`, for full marks.

## Question 2 (8 points):

**Learning Objectives**

**Purpose:** Students will practice the following skills:

- Implementing a function that works with node chains created using the Node ADT.

**Degree of Difficulty:** Easy.

**References:** You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes

**Restrictions:** This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Implement the function `check_chains()` . The interface for the function is:

```
def check_chains(chain1, chain2):
    """
    Purpose:
        Checks 2 node chains.
           If they are identical (the same objects),
                returns a string "same chain"
           If they are equal (same data values in the same order),
                returns a string "same values"
           Otherwise, returns a string "different starting at i"
                where i is an integer indicating the first different
                data value;
    Pre-conditions:
        :param chain1: a node-chain, possibly empty
        :param chain2: a node-chain, possibly empty
    Post-conditions:
        None
    Return:
        :return: a string
    """
```

On Moodle, you will find a *starter file* called `a5q2.py` containing the above interface documentation.

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

```
chain1 = N.node(1,
         N.node(1,
         N.node(9)))
chain2 = N.node(2,
         N.node(15))
chain3 = N.node(1,
         N.node(1,
         N.node(9)))
print(check_chains(chain1, chain1))
print(check_chains(chain1, chain2))
print(check_chains(chain1, chain3))
```

The output from the demonstration is as follows:

```
same chain
different starting at 0
same values
```

The integer value given in the second example is an offset. It's zero here because the first data value is different. If two node chains are different only in the second data value, then the offset would be 1. Finally, if there are more data values that are different, then only the offset of the first difference is mentioned.

## Questions

Answer the following questions about nodes, and node-chains:

1. Suppose we create a pair of node chains as follows:

   ```
   example1 = N.node(1, N.node(2, N.node(3)))
   example2 = N.node(1, N.node(2, N.node(3)))
   ```

   What happens when we call `example1 == example2`? Explain the result of this expression, in your own words.

2. Suppose you were writing test cases for a function `double_chain()` that doubles the numeric value of every node in the node chain. For example:

   ```
   example3 = double_chain(example1)
   example4 = N.node(2, N.node(4, N.node(6)))
   ```

   You don't have the code for this function, but assume it returns a node chain very similar to `example4`. How could you use `check_chains()` in test cases for `double_chain()`? To answer this question, give one test case for the (missing) function `to_chain()` that shows you understand how to use ~~`to_string()`~~ `check_chains()` for testing.

## What to Hand In

- A file named `a5q2.py` with the corrected definition of the function.
- Added: A file named `a5q2_answers.txt` with the answers to the questions above.

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

## Evaluation

- 4 marks: The function `check_chains()` works correctly
- 4 marks: Your answers to the questions demonstrate that you understand the value of `check_chains()` in testing.

**Solution:** Here's a model implementation of `check_chains()`.

```python
def check_chains(chain1, chain2):
    """
    Purpose:
        Checks 2 node chains.
           If they are identical (the same objects),
                returns a string "same chain"
           If they are equal (same data values in the same order),
                returns a string "same values"
           Otherwise, returns a string "different starting at i"
                where i is an integer indicating the first different
                data value;
    Pre-conditions:
        :param chain1: a node-chain, possibly empty
        :param chain2: a node-chain, possibly empty
    Post-conditions:
        None
    Return:
        :return: a string
    """

    # first check if both chains are identical
    if chain1 is chain2:
        return "same chain"

    # not identical, so check if they have the same data values
    # in the same order
    walker1 = chain1
    walker2 = chain2
    idx = 0

    # step through all the nodes until a difference is noted
    while walker1 is not None and walker2 is not None:
        if walker1.get_data() != walker2.get_data():
            return 'different starting at {}'.format(idx)
        walker1 = walker1.get_next()
        walker2 = walker2.get_next()
        idx += 1

    # if we get here, it's because one or both walkers is None
    if walker1 is not walker2:
        # if only one is None, then one chain is shorter, and therefore different
        return 'different starting at {}'.format(idx)
    else:
        # if they are both None, then all the data values are the same
        return 'same values'
```

The algorithm here is pretty straight-forward. One walker for each chain, taking a single step along each chain until a difference is spotted.

The key insight is the while loop. There are three ways to leave the while loop.

1. A difference before either walker reaches the end of a chain. Return with the index.
2. Reaching the end of one chain, but not the other. The loop condition is False, and the chains are different.
3. Reaching the end of both chains at the same time. The loop condition is False, and the chains are the same.

The problem is that we can't know which of the last two reasons is the case until wee look. That's the purpose of the if statement at the bottom.

Debugging a function like this is not straight-forward, though.

**Questions answered**

1. If we call `example1 == example2`, the answer is False. Python does not know how to look more carefully at the node chains, since nodes are not built-in. In this case, Python tells us only that they are different objects, stored at different addresses.
2. I would prefer to see what's in the chain, because it will help me understand the structure better.
3. We can sometimes use `check_chains()` to see if the node chain we get back from a function is exactly the same structure and data as another node chain:

```
input = N.node(1, N.node(2, N.node(3)))
expected = N.node(2, N.node(4, N.node(4)))

result = double_chains(input)
if check_chains(result, expected) != 'same chain':
    print('Error')
```

**Marking guidelines:**

- Give full marks if the function works.
- For the questions:
  1. Any answer talking about idea that the objects are stores in different locations, but nothing about the structure, gets full marks.
  2. Comparing a known node-chain to the result of `double_chains()`, for full marks.

## Question 3 (18 points):

**Note: the points for this question has been reduced, because the testing script was given. There is no longer a requirement to create a separate test script. Use the given one.**

**Learning Objectives**

**Purpose:** Students will practice the following skills:

- Working with node chains created using the Node ADT.

**Degree of Difficulty:** Easy to Moderate.

**References:** You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes

**Restrictions:** This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

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 `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!

(The rest of this page is blank so that each of the parts that follow can be presented on a single page.)

(a) (6 points) Implement and test the function `sumnc()` (the `nc` suggests node chain). The interface for the function is:

```
def sumnc(node_chain):
    """
    Purpose:
        Given a node chain with numeric data values, calculate
        the sum of the data values.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty, containing
                           numeric data values
    Post-condition:
            None
    Return
            :return: the sum of the data values in the node chain
    """
```

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

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

print('empty chain has the sum', sumnc(empty_chain))
print('chain has the sum', sumnc(chain))
```

The output from the demonstration is as follows:

```
empty chain has the sum 0
chain has the sum 6
```

(b) (6 points) Implement and test the function `count_in()`. The interface for the function is:

```
def count_in(node_chain, value):
    """
    Purpose:
        Counts the number of times a value appears in a node chain
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
        :param value: a data value
    Return:
        :return: The number times the value appears in the node chain
    """
```

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

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

print('empty chain has', count_in(empty_chain, 1), 'occurrences of the value 1')
print('chain has', count_in(chain, 1), 'occurrences of the value 1')
```

The output from the demonstration is as follows:

```
empty chain has 0 occurrences of the value 1
chain has 2 occurrences of the value 1
```

(c) (6 points) Implement and test the function `replace_in()`. The interface for the function is:

```
def replace_in(node_chain, target, replacement):
    """
    Purpose:
        Replaces each occurrence of the target value with the replacement
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
        :param target: a value that might appear in the node chain
        :param replacement: the value to replace the target
    Pre-conditions:
        Each occurrence of the target value in the chain is replaced with
        the replacement value.
    Return:
        None
    """
```

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

```
chain1 = N.node(1,
         N.node(1,
         N.node(9)))
chain2 = N.node(2,
         N.node(7,
         N.node(15)))
print('chain1 before:', to_string(chain1))
replace_in(chain1, 1, 10)
print('chain1 after:', to_string(chain1))

print('chain2 before:', to_string(chain2))
replace_in(chain2, 7, 1007)
print('chain2 after:', to_string(chain2))
```

The output from the demonstration is as follows:

```
chain1 before: [ 1 | *-]-->[ 1 | *-]-->[ 9 | / ]
chain1 after: [ 10 | *-]-->[ 10 | *-]-->[ 9 | / ]
chain2 before: [ 2 | *-]-->[ 7 | *-]-->[ 15 | / ]
chain2 after: [ 2 | *-]-->[ 1007 | *-]-->[ 15 | / ]
```

## What to Hand In

- A file named `a5q3.py` with the definitions of the three functions.
- ~~A file named `a5q3_testing.py` with a test script with test cases for these three functions.~~

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

## Evaluation

- 6 marks: Your function `sumnc()`:
  - Uses the Node ADT to sum the data values in the chain correctly.
  - Works on node-chains of any length.
  - If your function uses for-loops, Python lists, or the Python `sum()` function, you will get ZERO marks.
- 6 marks: Your function `count_in()`:
  - Uses the Node ADT to count the data values in the chain correctly.
  - Works on node-chains of any length.
  - If your function uses for-loops, or Python lists, you will get ZERO marks.
- 6 marks: Your function `replace_in()`:
  - Uses the Node ADT to replace the data values in the chain correctly.
  - Works on node-chains of any length.
  - Does not create any new nodes.
  - If your function uses for-loops, or Python lists, you will get ZERO marks.
- ~~9 marks: Your test cases for the three functions are good.~~
  - ~~Beginning with A5, test cases will be scrutinized more carefully, so make sure you think carefully about which cases you include.~~

**Solution:**

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

```python
def sumnc(node_chain):
    """
    Purpose:
        Given a node chain with numeric data values, calculate
        the sum of the data values.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty, containing
                           numeric data values
    Post-condition:
            None
    Return
            :return: the sum of the data values in the node chain
    """
    sum = 0
    walker = node_chain
    while walker is not None:
        sum += walker.get_data()
        walker = walker.get_next()

    return sum
```

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Give zero marks if the function makes use of Python lists, or `sum()`, or in any way evades the learning objective.

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

```
def count_in(node_chain, value):
    """
    Purpose:
        Counts the number of times a value appears in a node chain
    Pre-conditions:
        :param node_chain: a node chain, possibly empty
        :param value: a data value
    Return:
        :return: The number times the value appears in the node chain
    """
    count = 0
    walker = node_chain
    while walker is not None:
        if walker.get_data() == value:
            count += 1
        walker = walker.get_next()

    return count
```

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Give zero marks if the function makes use of Python lists, or in any way evades the learning objective.

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

```
def replace_in(node_chain, target, replacement):
    """
    Purpose:
        Replaces each occurrence of the target value with the replacement
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
        :param target: a value that might appear in the node chain
        :param replacement: the value to replace the target
    Pre-conditions:
        Each occurrence of the target value in the chain is replaced with
        the replacement value.
    Return:
        None
    """
    walker = node_chain
    while walker is not None:
        if walker.get_data() == target:
            walker.set_data(replacement)
        walker = walker.get_next()
```

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Give zero marks if the function makes use of Python lists, or in any way evades the learning objective.

## Question 4 (12 points):

**Learning Objectives**

**Purpose:** Students will practice the following skills:

- Working with node chains created using the Node ADT.

**Degree of Difficulty:** Moderate.

**References:** You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes

**Restrictions:** This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

In this question you'll write ~~four~~ two functions for node-chains that are a little more challenging. On Moodle, you can find a *starter file* called `a5q4.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 `a5q4_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

(a) (4 points) Implement and test the function `copync()`. The interface for the function is:

```
def copync(node_chain):
    """
    Purpose:
        creates a duplicate of the given node chain
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Post-conditions:
        None
    Return:
        :return: a new node chain, a node-for node copy
                 of the given one
    """
```

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

```
chain1 = N.node(1,
         N.node(1,
         N.node(9)))

chain2 = copync(chain1)
print(check_chains(chain1, chain2))
```

The output from the demonstration is as follows:

```
same values
```

(b) (4 points) Implement and test the function `double_up()`. The interface for the function is:

```
def double_up(node_chain):
    """
    Purpose:
        Modifies the node chain so that every node is doubled.
        E.g., given 1 -> 2 -> 3
              changed to 1 -> 1 -> 2 -> 2 -> 3 -> 3
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty

    Post-conditions:
        The chain is modified to have each node repeated once.
    Return:
        None
    """
```

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

```
chain1 = N.node(1,
         N.node(2,
         N.node(9)))

before_str = to_string(chain1)
double_up(chain1)
after_str = to_string(chain1)

print('before:', before_str)
print('after:', after_str)
```

The output from the demonstration is as follows:

```
before: [ 1 | *-]-->[ 2 | *-]-->[ 9 | / ]
after: [ 1 | *-]-->[ 1 | *-]-->[ 2 | *-]-->[ 2 | *-]-->[ 9 | *-]-->[ 9 | / ]
```

## What to Hand In

A file named `a5q4.py` with the definitions of the ~~three~~ two functions. Be sure to include your name, NSID, student number, and course number at the top of all documents.

## Evaluation

- 4 marks: Your function `copync()`:
  - Uses the Node ADT to create a copy of the chain correctly.
  - Works on node-chains of any length.
  - If your function uses for-loops, or Python lists, or Python functions `copy()` or `deep_copy()`, you will get ZERO marks.
- 4 marks: Your function `double_up()`:
  - Uses the Node ADT to create duplicate nodes in the chain correctly.
  - Works on node-chains of any length.
  - If your function uses for-loops, Python lists, or the Python `sum()` function, you will get ZERO marks.

**Solution:**

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

```
def copync(node_chain):
    """
    Purpose:
        creates a duplicate of the given node chain
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Post-conditions:
        None
    Return:
        :return: a new node chain, a node-for node copy
                 of the given one
    """

    # always have to check for None
    if node_chain is None:
        return None

    # a non-empty node-chain
    # we need to grab the first node in the copy, so we can return it
    duplicate = N.node(node_chain.get_data())

    # now walk along the original; for each node make a copy
    walker = node_chain
    dwalker = duplicate
    while walker.get_next() is not None:
        walker = walker.get_next()
        # make a new node with the right data
        dnode = N.node(walker.get_data())
        # connect the new node to the chain
        dwalker.set_next(dnode)
        # take a step along the duplicate chain
        dwalker = dnode

    return duplicate
```

The function walks along the given node-chain in the usual manner, checking if we have seen the current data value before.

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Give zero marks if the function makes use of Python lists, or in any way evades the learning objective.

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

```python
def double_up(node_chain):
    """
    Purpose:
        Modifies the node chain so that every node is doubled.
        E.g., given 1 -> 2 -> 3
              changed to 1 -> 1 -> 2 -> 2 -> 3 -> 3
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty

    Post-conditions:
        The chain is modified to have each node repeated once.
    Return:
        None
    """
    # a normal node chain walk
    walker = node_chain
    while walker is not None:
        # but here, create a new node, with the same value
        anode = N.node(walker.get_data(), walker.get_next())
        walker.set_next(anode)
        # since we added the new node, we have to take 2 steps!
        walker = walker.get_next().get_next()
```

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Give zero marks if the function makes use of Python lists, or in any way evades the learning objective.

## Question 5 (18 points):

**Learning Objectives**

**Purpose:** Students will practice the following skills:

- Working with node chains created using the Node ADT to implement slightly harder functionality.

**Degree of Difficulty:** Moderate to Tricky

**References:** You may wish to review the following:

- Chapter 3: References
- Chapter 12: Nodes

**Restrictions:** This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

In this question you'll implement merge sort for node chains! Recall, merge sort is a divide-and-conquer technique that uses recursion to sort a given sequence (in this case a node chain). A overview of the algorithm is given below.

```
Algorithm mergeSort(NC)
     # sorts node chain NC using merge sort
     # NC - a node chain of data items
     # return : new sorted node chain of NC

     if NC contains 0 or 1 data items:
          return NC

     # divide
     NC1 = first half of NC
     NC2 = second half of NC

     # recursively sort NC1 and NC2
     NC1 = mergeSort(NC1)
     NC2 = mergeSort(NC2)

     # conquer !!!
     NC = merge(NC1, NC2)

     return NC
```

In order to implement merge sort, you are going to write three functions that will make your job easier. On Moodle, you can find a *starter file* called `a5q5.py`, with all the functions and doc-strings in place, your job is to write the bodies of the functions. You will also find a test script named `a5q5_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 = N.node(3,
         N.node(1,
         N.node(4,
         N.node(1,
         N.node(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.

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

```
def merge(nc1, nc2):
    """
    Purpose:
        Combine the two sorted node-chains nc1 and nc2 into a single
        sorted node-chain.
    Pre-conditions:
        :param nc1: a node-chain, possibly empty,
        containing values sorted in ascending order.
        :param nc2: a node-chain, possibly empty,
        containing values sorted in ascending order.
    Post-condition:
        None
    Return:
        :return: a sorted node chain (nc) that contains the
        values from nc1 and nc2. If both node-chains are
        empty an empty node-chain will be returned.
    """
```

This one is tricky, as there are a few special cases.
A demonstration of the application of the function is as follows:

```
chain1 = N.node(1,
         N.node(1,
         N.node(9)))
chain2 = N.node(2,
         N.node(7,
         N.node(15)))
print('chain1 before:', to_string(chain1))
print('chain2 before:', to_string(chain2))
merged_chain = merge(chain1, chain2)
print('chain1 after:', to_string(chain1))
print('chain2 after:', to_string(chain2))
print('merged_chain after:\n', to_string(merged_chain))
```

The output from the demonstration is as follows:

```
chain1 before: [ 1 | *-]-->[ 1 | *-]-->[ 9 | / ]
chain2 before: [ 2 | *-]-->[ 7 | *-]-->[ 15 | / ]
chain1 after: [ 1 | *-]-->[ 1 | *-]-->[ 9 | / ]
chain2 after: [ 2 | *-]-->[ 7 | *-]-->[ 15 | / ]
merged_chain after:
[ 1 | *-]-->[ 1 | *-]-->[ 2 | *-]-->[ 7 | *-]-->[ 9 | *-]-->[ 15 | / ]
```

**Note:** There are two ways to solve this. The easier way is to create a chain of new nodes, using the data values in `nc1` and `nc2`. However, creating new nodes takes a small amount of time, and memory. The more efficient way to implement this function is to re-use the given nodes, and just make the arrows point differently. Only do this if (1), you have the less efficient way working, and (2) if you want a programming challenge.

(c) (3 points) Implement the function `merge_sort()`. The interface for the function is:

```
def merge_sort(node_chain):
    """
    Purpose:
        Sorts the given node chain in ascending order using the
        merge sort algorithm.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty,
        containing only numbers
    Post-condition:
        the original node_chain may be modified and will likely
        not contain all the original elements
    Return
        :return: the node-chain sorted in ascending order.
        Ex: 45->1->21->5. Becomes 1->5->21->45
    """
```

This one might be a little difficult, since it uses recursion.

**22/06/2021 Note:** The post-conditions in the above are corrected to reflect the starter file given for the assignment.

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

```
chain = N.node(10,
        N.node(9,
        N.node(12,
        N.node(7,
        N.node(11,
        N.node(8))))))
print('chain before:\n', to_string(chain))
sorted_node_chain = merge_sort(chain)
print('merge_sort results:\n', to_string(sorted_node_chain))
```

The output from the demonstration is as follows:

```
chain before:
[ 10 | *-]-->[ 9 | *-]-->[ 12 | *-]-->[ 7 | *-]-->[ 11 | *-]-->[ 8 | / ]
merge_sort results:
[ 7 | *-]-->[ 8 | *-]-->[ 9 | *-]-->[ 10 | *-]-->[ 11 | *-]-->[ 12 | / ]
```

(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 `a5q5.py` with the definition of your functions. Be sure to include your name, NSID, student number, and course number 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 `merge()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT to create a new node-chain, when given two node chains already in ascending order. The resulting node-chain is also sorted in ascending order (node-chain only contains numbers).
  - Works on node-chains of any length.
- 3 marks: Your function `merge_sort()`:
  - Does not violate the Node ADT.
  - Uses the functions above to sort a given node-chain in ascending order.
  - 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
    """
    # deal with small chains as special cases
    if node_chain is None:
        return None, None
    elif node_chain.get_next() is None:
        return None, node_chain
    else:
        # the chain is 2 nodes long or longer
        # technique: Two walkers, one hops twice as fast
        # when the fast walker gets to the end,
        # the slow one is half-way down
        walker = node_chain
        half_speed = node_chain
        prev = None
        while walker is not None:
            walker = walker.get_next()
            # hop a second time, but carefully!
            if walker is not None:
                walker = walker.get_next()
                prev = half_speed
                half_speed = half_speed.get_next()

        # split the chain right infront of the slow walker
        prev.set_next(None)

        return node_chain, half_speed
```

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..
- Deduct 2 marks if Python lists were used somehow.

(b) Here's an implementation of `merge()` that would be acceptable (the doc-string is omitted to save a bit of space on this page):

```
def merge_v1(nc1, nc2):
    # first, check for empty node-chains
    if nc1 is None:
        # could copy the other node-chain, but why bother?
        return nc2
    elif nc2 is None:
        return nc1
    # neither is None, so look at the data value to see which goes first
    elif nc1.get_data() < nc2.get_data():
        result = N.node(nc1.get_data())
        nc1 = nc1.get_next()
    else:
        result = N.node(nc2.get_data())
        nc2 = nc2.get_next()

    # result refers to the first node in the merged node chain
    # need a walker to make the appropriate connections
    rwalker = result
    while nc1 is not None and nc2 is not None:
        # look for the smaller of the two data values
        # advance only the one with the smaller value
        if nc1.get_data() < nc2.get_data():
            new_data = nc1.get_data()
            nc1 = nc1.get_next()
        else:
            new_data = nc2.get_data()
            nc2 = nc2.get_next()

        # create a new node, and advance the walker
        rwalker.set_next(N.node(new_data))
        rwalker = rwalker.get_next()

    # here, we've reached the end of one or both of the original chains
    if nc1 is None:
        # could copy the other node-chain, but why bother?
        rwalker.set_next(nc2)
    else:
        # could copy the other node-chain, but why bother?
        rwalker.set_next(nc1)

    # finally, return the result
    return result
```

There are 2 special cases: when one or both the node chains are empty. The code looks to see which one of the node chains has the smallest first value, and keeps it. After that, a while-loop walks along, looking for the smallest value, moving it to the merged result. The while-loop only advances one of the node-chains.

Here's an alternate implementation of `merge()` that would be acceptable (the doc-string is omitted to save a bit of space on this page):

```python
def merge_v2(nc1, nc2):
    # first, check for empty node-chains
    if nc1 is None:
        return nc2
    elif nc2 is None:
        return nc1
    # neither is None, so look at the data value to see which goes first
    elif nc1.get_data() < nc2.get_data():
        result = nc1
        nc1 = nc1.get_next()
    else:
        result = nc2
        nc2 = nc2.get_next()

    # result refers to the first node in the merged node chain
    # need a walker to make the appropriate connections
    rwalker = result
    while nc1 is not None and nc2 is not None:
        # look for the smaller of the two data values
        # advance only the one with the smaller value
        if nc1.get_data() < nc2.get_data():
            rwalker.set_next(nc1)
            nc1 = nc1.get_next()
        else:
            rwalker.set_next(nc2)
            nc2 = nc2.get_next()

        # advance the walker
        rwalker = rwalker.get_next()

    # here, we've reached the end of one or both of the original chains
    if nc1 is None:
        rwalker.set_next(nc2)
    else:
        rwalker.set_next(nc1)

    # finally, return the result
    return result
```

This is very similar to the previous, but this one's doesn't create any new nodes. Instead, it reuses all the nodes in the given node-chains.

Here's an alternate implementation of `merge()` that would be acceptable (the doc-string is omitted to save a bit of space on this page):

```
def merge_rec(nc1, nc2):
    # if one is empty, return the other one
    if nc1 is None:
        return nc2
    elif nc2 is None:
        return nc1
    else:
        # both not empty
        # grab the data values
        first_val = nc1.get_data()
        second_val = nc2.get_data()

        # check which comes first
        # return a new node chain with the smaller value in front
        # of all the others
        if first_val <= second_val:
            return N.node(first_val, merge_rec(nc1.get_next(), nc2))
        else:
            return N.node(second_val, merge_rec(nc1, nc2.get_next()))
```

This one is a recursive version. It turns out to be the shortest solution of the bunch. I think it is fair to say that it is simpler than the others as well. The same two special cases as base cases. But the recursive case does a lot in a few lines. The smallest value is used in the new node, and only one of the two walkers is advanced.

**Marking guidelines:**

- For full marks, the function must walk along the chain.
- Deduct 2 marks if Python lists were used somehow.

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

```
def merge_sort(node_chain):
    """
    Purpose:
        Sorts the given node chain in ascending order using the
        merge sort algorithm.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty,
        containing only numbers
    Post-condition:
        the original node_chain may be modified and will likely
        not contain all the original elements
    Return
        :return: the node-chain sorted in ascending order.
        Ex: 45->1->21->5. Becomes 1->5->21->45
    """

    if node_chain is None:
        return None
    elif node_chain.get_next() is None:
        return node_chain
    else:
        nc1, nc2 = split_chain(node_chain)
        snc1 = merge_sort(nc1)
        snc2 = merge_sort(nc2)
        return merge(snc1, snc2)
```

This one should be easy if the other two functions are implemented correctly. Note students should have use the `split_chain()` and `merge()` functions for the "divide" and "conquer" portions of merge sort.

**Marking guidelines:**

- For full marks, the functions were implement correctly using the Node ADT only.
- Deduct 1 marks if Python lists were used somehow.

(d) Programming style. Students have been advised on good style (see Chapter 14 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.

Note for markers, only consider the `split_chain()` and `merge()` functions here.

**Marking guidelines:**

- Deduct 1 marks if variable names are not appropriate.
- Deduct 2 marks if there are no comments at all, or if there is excessive commenting.