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
Winter 2019
Principles of Computer Science

Assignment 5

Node chains

Date Due: February 15, 2019, 7pm **Total Marks: 53**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

Question 0 (6 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

- 6 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.

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.

When you get this function working again, use it to help you test and debug the rest of the questions in this assignment.

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

Question 2 (24 points):

Purpose: To practice working with node chains created using the Node ADT.

Degree of Difficulty: Easy to Moderate.

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!

Use `to_string()` from Q1 to help you test and debug your functions.

(a) (6 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) (6 points) Implement the function `contains_duplicates()`. The interface for the function is:

```
def contains_duplicates(node_chain):
    """
    Purpose:
        Returns whether or not the given node_chain contains one
        or more duplicate data values.
    Pre-conditions:
        :param node_chain: a node-chain, possibly empty
    Return:
        :return: True if duplicate data value(s) were found,
        False otherwise
    """
```

For this question, you are NOT allowed to use a List to store data values, instead consider using two walkers and a nested loop. This operation should simply return True if we find ANY duplicate values, or False otherwise. Pseudocode for the algorithm follows on the next page.

```
start a walker to walk along the node chain
while walker has not reached the end
   start a checker to walk from walker to the end of the chain
   while checker has not reached the end
      if checker's data value is the same as walker's, return True
if walker reached the end of the chain, return False
```

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

```
chain1 = node.create(1,
         node.create(2,
         node.create(3,
         node.create(4,
         node.create(5)))))
print('Duplicates?', contains_duplicates(chain1))
chain2 = node.create(1,
         node.create(2,
         node.create(3,
         node.create(4,
         node.create(1)))))
print('Duplicates?', contains_duplicates(chain2))
```

The output from the demonstration is as follows:

```
Duplicates? False
Duplicates? True
```

(c) (6 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 $n-1$, 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 $n-1$, 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) (6 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 three functions. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 6 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.
- 6 marks: Your function `contains_duplicates()`:
  - Does not violate the Node ADT.
  - Uses the Node ADT (and does NOT use lists) to check if the node-chain contains any duplicate data values.
  - Works on node-chains of any length.
- 6 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.
- 6 marks: Overall, you used good programming style, including:
  - Good variable names
  - Appropriate internal comments (outside of the given doc-strings)

Question 3 (18 points):

Purpose: To practice working with node chains created using the Node ADT to implement slightly harder functionality.

Degree of Difficulty: Moderate to Tricky

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 `a5q3.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 `a5q3_testing.py`. It has a bunch of test cases pre-written for you. Read it carefully!

Use `to_string()` from Q1 to help you test and debug your functions.

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

```python
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 `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 = node.create(1,
         node.create(1,
         node.create(9)))
chain2 = node.create(2,
         node.create(7,
         node.create(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 | / ]
```

(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 node-chain will be sorted in ascending order.
    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.
A demonstration of the application of the function is as follows:

```
chain = node.create(10,
         node.create(9,
         node.create(12,
         node.create(7,
         node.create(11,
         node.create(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 `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 `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)