DIT/FT/2B/11

DSAA CA2 Final report

Done by Yeo Sheen Hern (1902257) & Kaeden Tan (1935529)

1. Operation of Application

1.1. Selection

Upon launching the program, a prompt (as seen in appendix A Figure 1) will appear to select from 3 options, Evaluating from expression, Evaluating from a file, and exiting the program. The user can select an option by entering 1, 2, or 3 in the CLI and any other inputs are rejected.

1.2. Options

1.2.1. Evaluate expression (Option 1)

When selected, the program will prompt the user to input an expression (as seen in appendix A Figure 2). If an invalid expression is given, the error will be printed in the terminal and the prompt will come up again. Once a valid expression is input, a prompt will come up to choose different tree traversals to print from. Upon a valid input, the tree traversal order as well as the evaluation of the expression will be printed. All evaluated expressions will be saved in a text file called input_history.txt. Thereafter the user has to press enter to go back to the selection menu.

1.2.2. Evaluate from file (Option 2)

When selected, the program will prompt the user to input the name of the input file and output file (as seen in appendix A Figure 2). If an invalid input is given, the error will be printed in the terminal and the prompt will come up again. The output file only allows for .txt files. The program will then prompt the user for the type of sort they want, the order of the sort (asc/desc), their preferred sorting criterion, and if they chose the 2nd option in the preferred sorting criterion, will be prompted to input the value they would like the expressions to be sorted by. This input is limited to only valid operators and numbers and will sort the expression based on the count of the sorting criteria. The program will then print the results. Thereafter the user has to press enter to go back to the selection menu.

1.3. User Guidelines

Supported Expressions: Operators Plus '+', Minus (unary and binary) '-', Multiply '*', Divide '/', Exponent '**', Modulo '%', Parentheses '()'. Expressions **DO NOT** have to be parenthesized.

2. Implementation with Object-Oriented Programming

This section will cover the classes we use for the application, their use cases and how they interact together. It will also delve into our reasons for structuring our code this way with regards to Object-Oriented Programming Principles, mainly regarding the changes since our interim report.

2.1. Classes & Functions unchanged from Interim report.

2.1.1. Stack and BinaryTree collection classes

The Stack and BinaryTree remained largely unchanged and still serve the same purpose, as dependencies for the Expression class. They aim to encapsulate key data structures that we use in our application.

2.1.2. Expression class

The Expression class is also the same as before, it is made to build the tree, parse it and store the expression string and value of the expression. We have decided to implement the building of the tree using the shunting yard algorithm and the evaluation by recursing through the entire tree. It serves to abstract everything to do with any particular expression into an Expression object.

2.1.3. Sort_expression function

The sort_expression function works the same as planned, it accepts a list containing expression objects with a variable indicating ascending or descending. The function uses the merge sort algorithm to sort the list.

2.2. Added and changed Classes & Functions

2.2.1. Validation function

We originally planned to first validate the expression here before passing it to the Expression Class, however, we forwent the idea and decided to validate it in our Tokenizer class. How we did this is covered in section 2.2.2 Tokenizer Class.

2.2.2. Tokenizer Class

The Tokenizer class was created to tokenize the expression in order to build the parse tree easily. Using the concept of lexer analysis, we created TokenTypes (Inheriting Enum) such as operators, numbers, and parenthesis. By scanning through each character of the string, we identified the corresponding operator or number, yielded it as a generator, then converted it into a list for use. In the case where consecutive characters form 1 token (e.g. Floats, exponents), the tokenizer would handle it in a separate function, looking at the following characters to determine the token to be generated. For example, if the Tokenizer finds a number, it will look at the following characters, if it is a number or decimal it will add it to the token and continue until the next character is no longer a number.

2.2.2.1. Validation in Tokenizer

The tokenizer handles all validation while it scans through the string. If an invalid character, multiple decimal points in a string of numbers, or an empty expression is passed to the tokenizer, it will raise an exception that is handled in the interface.

2.2.3. SortedList class

While we have already implemented a merge sort, we also implemented a sorted list to demonstrate inheritance in the Expression class. This class is implemented to store the head node of a linked list as well as the current node it is on in order to iterate through the linked list. The node that is stored is an expression object while the expression class inherits the next_node variable from the Node class which points to the next node in the linked list. Every time something is inserted it will iterate through the entire list until it finds a place where it is larger or lesser than the next node depending on whether it is an ascending or descending sort.

2.2.4. CLInterface Class

The CLInterface class is created so we can better structure our code by abstracting the Interface which handles all the prints and Interface logic into a class. The CLInterface class is used in the main program loop mainly for menu selection, making both our main program loop, and CLInterface cleaner and more readable.

2.3. Challenges and Solutions from Implementation

2.3.1. Tokenization

Initially, we had planned on using python's in-built tokenize class for the tokenization of our user input expressions, then post-processing the tokens generated for our use-case. However, we realised that this was much more difficult than expected, as the tokens generated were uncustomizable, and there might not even be a way to post-process them into our desired format. Hence, we did some research on how human "languages" can be more easily parsed by computers.

We then came across the concept of Lexers and Parsers, commonly used for compiling programming languages. These lexers can be built to follow their own sets of grammatical rules and identifiers, making them very suitable for tokenizing mathematical expressions for our use-case. By building our own lexer, we are also able to more easily validate the human input expression with our lexer, and hence no longer need to use the validate_input helper class that we had originally planned. Therefore, we decided to use this approach to build our Tokenizer class (Lexer) to make the tokens to be used in our parse_tree function for building our expression tree (Parser).

2.3.2. Validation of Expression

Even though we can easily validate user input for only valid Tokens to be generated, we still need to validate the Expression itself to make sure that it follows mathematical rules. There were considerations on whether we should validate the expression by performing checks on the tokens or by just raising exceptions when parsing the tokens and building the tree itself. The solution wasn't clear, so as always, we decided to do more research.

Eventually, we came across a concept of having "algorithm states" in our shunting yard algorithm. This included 2 states of "want operator" and "want operand", where after parsing each token, we can set the state to expect an operator/operand next, raising an exception whenever the next token doesn't match the expected type. This allows us to easily catch invalid mathematical expressions in the parsing stage that managed to be tokenized by our lexer.

2.3.3. Unary Operators

Finally, the implementation of unary operators was tricky, especially to incorporate it with the shunting yard algorithm as the algorithm logic was already pretty complicated. We also had to consider whether it was better to implement separate TokenTypes to handle unary/binary minuses. Eventually, we found the criteria for determining whether a minus was unary or binary by looking what precedes it. Since it is hard to access the previous token in the lexer, we decided to detect unary minuses in our shunting yard algorithm. Upon encountering one, we reversed the sign of the next number token instead of adding it to the operator stack.

2.4. Key takeaways and learning points

We learnt a lot in this project. As we have not worked data structures outside of iterables and key value pair data structures, researching on how the whole thing worked was enriching in and of itself. One especially interesting point of learning was lexers. The way it tokenizes a string with a predefined dictionary and how this concept can be extended to compiling programming languages was impressive. Also, we learnt about algorithms like the shunting yard algorithm. The way the tree was built utilising a stack and the checking of operator precedence felt very novel when we first saw it.

We also learnt a lot about time complexities. Discussing the time complexities of various sorts and their pros and cons was something that we did not see ourselves doing, and we really felt like we learnt a lot about data structures and algorithms that would be helpful especially since both of us plan on furthering our studies in Computer Science.

Finally by working in a team, we learnt valuable lessons about project development, source control and Object-Oriented Programming. How to plan meetings, discuss project structure and learning about best practices with source control is something we are sure will help a lot in the future.

3. Advanced Features

Implemented Advanced Features include:

3.1. Evaluate without being parenthesized (Allow unary minus)

3.2. Supports Modulo Operator

3.3. Choice between Sorting Ascending/Descending Order

3.4. Extra Sorting Criteria (number of a certain operator/operand)

3.5. 3 ways of printing parse tree

3.6. Save evaluated expressions - Choice between mergesort and sortedlist

4. Role of members

Here are our individual responsibilities in a table format:

Sheen Hern	Kaeden
Tokenizer	Binary & Stack Data Structure
Expression (Parsing Tree Logic, Validation)	Sorting Expressions (Sorted list)
Expression (Evaluation)	Sorting Expressions (Merge sort)
Expression (Printing of Expression Tree)	Expression (Overloading for Sorting)
Validation of User Input (In tokenizing)	CLInterface (Printing & writing files)

Appendix (Terminal prints)

Figure 1. Main menu when starting

```
******************************************************************
* ST107 DSAA: Expression Evaluator & Sorter                      *
*----------------------------------------------------------------*
*  - Done by: Yeo Sheen Hern (1902257) & Kaedan Tan (1935529)    *
*  - Class DIT/FT/2B/11                                          *
******************************************************************


Please select your choice ('1','2','3')
   1. Evaluate expression
   2. Sort expressions
   3. Exit
Enter choice: 
```

Figure 2. Option full process

```
Please select your choice ('1','2','3')
   1. Evaluate expression
   2. Sort expressions
   3. Exit
Enter choice: 1

Please enter the expression you want to evaluate:
1+1**8-(123+10)**-1

Please select your choice ('1','2','3')
   1. Print Preorder
   2. Print Inorder
   3. Print Postorder
Enter choice: 2
_ _ 1.0
_ +
_ _ _ 1.0
_ _ **
_ _ _ 8.0
-
_ _ _ 123.0
_ _ +
_ _ _ 10.0
_ **
_ _ -1.0

Expression evaluates to:
1.992
Press enter to continue... 
```

Figure 3. Option 2 full process

```
Please select your choice ('1','2','3')
   1. Evaluate expression
   2. Sort expressions
   3. Exit
Enter choice: 2
Please enter input file: input.txt
Please enter output file: output.txt

Please select your choice ('1','2')
   1. Sorted list
   2. Merge sort
Enter choice: 2

Please select your choice ('1','2')
   1. Sort ascending
   2. Sort descending
Enter choice: 2

Please select your choice ('1','2')
   1. Sort by value then length
   2. Sort by count of user input
Enter choice: 2

Please enter the value (operator/number) you want to sort by:
1


>>>Evaluation and sorting started:



*** Expressions with a total of 10 1.0
(((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1)==>10.000


*** Expressions with a total of 2 1.0
((1+2)+(3+(3+1)))==>10.000


*** Expressions with a total of 1 1.0
(((1+2)+3)*4)==>24.000
((1+2)+(3+4))==>10.000
((1+2)+3)==>6.000


*** Expressions with a total of 0 1.0
((10+(10+(10+10)))+(10+10))==>60.000
(10+(20+30))==>60.000
((11.07+25.5)-10)==>26.570
(2+(2+2))==>6.000
((-500+(4*3.14))/(2**3))==>-60.930



>>>Evaluation and sorting completed!
Press enter to continue...
```

Appendix (Source Code Listings)

Expressions Class written by Sheen Hern

(excluding overloading)

```
from helpers import Tokenizer, TokenType

from collection import Stack, BinaryTree


class Node:

    # Constructor

    def __init__(self):

        self.nextNode = None


class Expression(Node):

    def __init__(self, exp_str):

        super().__init__()

        self.__exp_str = exp_str.replace(" ",  '')

        self.__tokens = self.tokenize_exp()

        self.__tree_root = None

        self.val = None

        self.sort_value = None


    # Tokenize Expression with our tokenizer

    def tokenize_exp(self):
```

```
        tokenizer = Tokenizer(self.__exp_str)

        tokens = tokenizer.generate_tokens()

        # return tokens

        return list(tokens)


    # Parse tokens into tree with shunting-yard algorithm

    def parse_tree(self):

        # Stacks to hold operators/operands for alg

        operator_stack = Stack()

        node_stack = Stack()


        prev_token = None

        # Alg states Expect Operand (0), Expect Operator (1), for catching invalid
expressions

        alg_state = 0


        for i, token in enumerate(self.__tokens):

            # Push Operands into operand_stack, operators to operator_stack

            if token.type == TokenType.LPAREN:

                if alg_state != 0:

                    raise Exception(f"Expected Operand after {prev_token}")

                operator_stack.push(token)

            elif token.type == TokenType.NUMBER:

                if alg_state != 0:

                    raise Exception(f"Expected Operand after {prev_token}")

                node_stack.push(token)

                alg_state = 1

            # For operators
```

```
            elif token.precedence > 0:
                # If unary minus, reverse sign of next number
                if (token.type == TokenType.MINUS) and (prev_token == None or (prev_token.type != TokenType.NUMBER and prev_token.type != TokenType.RPAREN)):
                    if alg_state != 0:
                        raise Exception(f"Expected Operand after {prev_token}")
                    for j in range(i, len(self.__tokens)):
                        if self.__tokens[j].type == TokenType.NUMBER:
                            self.__tokens[j].value = -self.__tokens[j].value
                            break
                        # No number after unary minus
                        elif j == len(self.__tokens)-1:
                            raise Exception(f"No number after unary minus")
                else:
                    if alg_state != 1:
                        raise Exception(f"Expected Operator after {prev_token}")

                    # If lower or equal precendence, also handles exponent (evals right to left)
                    while (not operator_stack.isEmpty() and operator_stack.get().type != TokenType.LPAREN
                        and ((token.type != TokenType.EXPONENT and operator_stack.get().precedence >= token.precedence)
                                or (token.type is TokenType.EXPONENT and operator_stack.get().precedence > token.precedence) )
                    ):
                        operator_stack, node_stack = self.make_subtree(operator_stack, node_stack)

                    # Push currrent token to operator stack
```

```
                    operator_stack.push(token)


                    alg_state = 0

            # Handle Parenthesis

            elif token.type == TokenType.RPAREN:

                if alg_state != 1:

                    raise Exception(f"Expected Operator after {prev_token}")

                # Pop all until LPAREN found

                while (not operator_stack.isEmpty() and operator_stack.get().type !=
TokenType.LPAREN):

                    operator_stack, node_stack = self.make_subtree(operator_stack,
node_stack)


                # Remove LPAREN

                operator_stack.pop()


            prev_token = token


        # Empty and build/connect rest of the tree after finishing all tokens

        while (not operator_stack.isEmpty()):

            operator_stack, node_stack = self.make_subtree(operator_stack,
node_stack)


        # Return built expression tree

        self.__tree_root = node_stack.get()

        # print('DONE TREE: ' + str(self.__tree_root))

        self.val = self.evaluate(self.__tree_root)

        return
```

```
# Makes subtree from stacks
def make_subtree(self, operator_stack, node_stack):
    # Parent node (top operator from stack)
    parent = operator_stack.pop()

    # Tree cannot have parentheses
    if parent.type == TokenType.LPAREN or parent.type == TokenType.RPAREN:
        raise Exception("Parentheses are mismatched.")

    # Get top operands from stack
    right = node_stack.pop()
    left = node_stack.pop()

    # Make sure no NoneTypes for subtree
    if parent == None or right == None or left == None:
        raise Exception("Ending with an operator")

    # Add subnodes to tree
    sub_tree = BinaryTree(parent, left, right)

    # Append to whole tree (node stack)
    node_stack.push(sub_tree)

    return operator_stack, node_stack

# Evaluate expression
def evaluate(self, binary_tree_node):
    # Empty Tree
```

```
        if binary_tree_node is None:
            return 0

        # Is a leaf node (Reached bottom)
        if type(binary_tree_node) != BinaryTree:
            return binary_tree_node.value

        left_sum = self.evaluate(binary_tree_node.get_left_tree())
        right_sum = self.evaluate(binary_tree_node.get_right_tree())

        # Apply operations
        if binary_tree_node.get_key().type == TokenType.PLUS:
            return left_sum + right_sum
        elif binary_tree_node.get_key().type == TokenType.MINUS:
            return left_sum - right_sum
        elif binary_tree_node.get_key().type == TokenType.MULTIPLY:
            return left_sum * right_sum
        elif binary_tree_node.get_key().type == TokenType.DIVIDE:
            return left_sum / right_sum
        elif binary_tree_node.get_key().type == TokenType.MODULO:
            return left_sum % right_sum
        elif binary_tree_node.get_key().type == TokenType.EXPONENT:
            return left_sum ** right_sum

    # Recursively prints the various tree traversals (Preorder, Postorder, Inorder)
    def print_preorder(self, tree=True, depth=0):
        if tree is True:
            tree = self.__tree_root
```

```
        if tree != None:
            if type(tree) != BinaryTree:
                # print(('-') * depth + str(tree.value))
                print(('- ') * depth + str(tree))
            else:
                print(('- ') * depth + str(tree.get_key()))
                self.print_preorder(tree.get_left_tree(), depth+1)
                self.print_preorder(tree.get_right_tree(), depth+1)

    def print_postorder(self, tree=True, depth=0):
        if tree is True:
            tree = self.__tree_root
        if tree != None:
            if type(tree) != BinaryTree:
                print(('- ') * depth + str(tree.value))
            else:
                self.print_postorder(tree.get_left_tree(), depth+1)
                self.print_postorder(tree.get_right_tree(), depth+1)
                print(('- ') * depth + str(tree.get_key()))

    def print_inorder(self, tree=True, depth=0):
        if tree is True:
            tree = self.__tree_root
        if tree != None:
            if type(tree) != BinaryTree:
                print(('- ') * depth + str(tree.value))
            else:
                self.print_inorder(tree.get_left_tree(), depth+1)
```

```
            print(('- ') * depth + str(tree.get_key()))

            self.print_inorder(tree.get_right_tree(), depth+1)


    # helping methods

    def set_sort_value(self, value):

        self.sort_value = value

        return


    def sort_value_count(self):

        return self.__tokens.count(self.sort_value)


    # Overloading operators

    def __lt__(self, other):

        if self.sort_value == None:

            if self.val != other.val:

                return self.val < other.val

            elif len(str(self)) != len(str(other)):

                return len(str(self)) < len(str(other))

        else:

            if self.__tokens.count(self.sort_value) !=
other.__tokens.count(self.sort_value):

                return self.__tokens.count(self.sort_value) <
other.__tokens.count(self.sort_value)

            elif self.val != other.val:

                return self.val < other.val

            elif len(str(self)) != len(str(other)):

                return len(str(self)) < len(str(other))
```

```
    def __gt__(self, other):

        if self.sort_value == None:

            if self.val != other.val:

                return self.val > other.val

            elif len(str(self)) != len(str(other)):

                return len(str(self)) > len(str(other))

        else:

            if self.__tokens.count(self.sort_value) !=
other.__tokens.count(self.sort_value):

                return self.__tokens.count(self.sort_value) >
other.__tokens.count(self.sort_value)

            elif self.val != other.val:

                return self.val > other.val

            elif len(str(self)) != len(str(other)):

                return len(str(self)) > len(str(other))



    def __str__(self):

        return self.__exp_str
```

Tokenizer Class written by Sheen Hern

```
from enum import Enum

from dataclasses import dataclass


# Tokenizer's Token DataClass

class TokenType(Enum):

    NUMBER = 0

    PLUS = 1

    MINUS = 2

    MULTIPLY = 3

    DIVIDE = 4

    MODULO = 5

    EXPONENT = 6

    LPAREN = 7

    RPAREN = 8


@dataclass

class Token:

    type: TokenType

    value: any = None

    rep: str = None

    precedence: int = None


    def __post_init__(self):

        precedences = {

            TokenType.NUMBER : None,

            TokenType.LPAREN : None,
```

```
            TokenType.RPAREN : 0,
            TokenType.PLUS : 1,
            TokenType.MINUS : 1,
            TokenType.MULTIPLY : 2,
            TokenType.DIVIDE : 2,
            TokenType.MODULO : 2,
            TokenType.EXPONENT : 3
        }
        self.precedence = precedences[self.type]


    def __repr__(self):
        representations = {
            TokenType.LPAREN : '(',
            TokenType.RPAREN : ')',
            TokenType.PLUS : '+',
            TokenType.MINUS : '-',
            TokenType.MULTIPLY : '*',
            TokenType.DIVIDE : '/',
            TokenType.MODULO : '%',
            TokenType.EXPONENT : '**'
        }
        return str(self.value) if self.value != None else representations[self.type]


# Tokenizer
WHITESPACE = ' \n\t'
DIGITS = '0123456789'
```

```python
class Tokenizer:
    def __init__(self, text):
        if text == "":
            raise Exception("Please input an expression")
        # Iteration of text
        self.text = iter(text)
        self.advance()

    # Advance to next character if avail
    def advance(self):
        try:
            self.current_char = next(self.text)
        except StopIteration:
            self.current_char = None

    def generate_tokens(self):
        while self.current_char != None:
            # Ignore whitespaces
            if self.current_char in WHITESPACE:
                self.advance()
            # Tokenize valid operands/operators
            elif self.current_char == '.' or self.current_char in DIGITS:
                # Handle numbers (many digits/decimal pts)
                yield self.generate_number()
            elif self.current_char == '+':
                self.advance()
                yield Token(TokenType.PLUS)
```

```python
        elif self.current_char == '-':
            self.advance()
            yield Token(TokenType.MINUS)
        elif self.current_char == '*':
            yield self.generate_asterisk()
        elif self.current_char == '/':
            self.advance()
            yield Token(TokenType.DIVIDE)
        elif self.current_char == '%':
            self.advance()
            yield Token(TokenType.MODULO)
        elif self.current_char == '(':
            self.advance()
            yield Token(TokenType.LPAREN)
        elif self.current_char == ')':
            self.advance()
            yield Token(TokenType.RPAREN)
        # Except all other characters
        else:
            raise Exception(f"Illegal character '{self.current_char}'")

# Generates numbers from each character (digit/decimal)
def generate_number(self):
    # TO TEST: try ".0.2", does it work with 2 decimals, starting with dec?
    decimal_count = 0
    number_str = ''

    # While still digits/decimals
```

```
        while self.current_char != None and (self.current_char == '.' or self.current_char in DIGITS):
            # No more than 1 decimal in number
            if self.current_char == '.':
                decimal_count += 1
                if decimal_count > 1:
                    raise Exception(f"More than 1 decimal in number")

            # Adds digits/decimals to number_str
            number_str += self.current_char
            self.advance()

        # Formats numbers starting with . appropriately (.12 -> 0.12)
        if number_str.startswith('.'):
            number_str = '0' + number_str

        # Formats numbers ending with . appropriately (12. -> 12)
        if number_str.endswith('.'):
            number_str += '0'

        return Token(TokenType.NUMBER, float(number_str))

    # Generates either MULTIPLY or EXPONENT Token
    def generate_asterisk(self):
        asterisk_count = 1
        self.advance()

        # Loop when still *
```

```
while self.current_char != None and (self.current_char == '*'):

    asterisk_count += 1

    # Cannot have more than 2 * in a row, invalid expression

    if asterisk_count > 2:

        break


    self.advance()


# Return respective Tokens

if asterisk_count == 1:

    return Token(TokenType.MULTIPLY)


return Token(TokenType.EXPONENT)
```

BinaryTree Class written by Kaeden

```
class BinaryTree:

    def __init__(self,key, leftTree = None, rightTree = None):

        self.key = key

        self.leftTree = leftTree

        self.rightTree = rightTree


    def set_key(self, key):

        self.key = key


    def get_key(self):

        return self.key


    def get_left_tree(self):

        return self.leftTree


    def get_right_tree(self):

        return self.rightTree


    def insert_left(self, key):

        if self.leftTree == None:

            self.leftTree = BinaryTree(key)

        else:

            t =BinaryTree(key)

            self.leftTree , t.leftTree = t, self.leftTree


    def insert_right(self, key):
```

```
        if self.rightTree == None:

            self.rightTree = BinaryTree(key)

        else:

            t =BinaryTree(key)

            self.rightTree , t.rightTree = t, self.rightTree
```

Stack Class written by Kaeden

```
class Stack:

    def __init__(self):

        self.stack_list = []


    def push(self, value):

        self.stack_list.append(value)


    def pop(self):

        if self.isEmpty():

            return None

        return self.stack_list.pop(-1)


    def get(self):

        if self.isEmpty():

            return None

        return self.stack_list[-1]


    def isEmpty(self):

        if len(self.stack_list) == 0:

            return True

        return False




    def __str__(self):

        print(str(self.stack_list))
```

SortedList Class written by Kaeden

```python
class SortedList:

    def __init__(self, ascending_check):

        self.headNode = None

        self.currentNode = None

        self.length = 0

        self.ascending_check = ascending_check


    def __appendToHead(self, newNode):

        oldHeadNode = self.headNode

        self.headNode = newNode

        self.headNode.nextNode = oldHeadNode


    def insert(self, newNode):

        self.length += 1

        # If list is currently empty

        if self.headNode == None:

            self.headNode = newNode

            return

        if self.ascending_check == '1':

            # Check if it is going to be new head

            if newNode < self.headNode:

                self.__appendToHead(newNode)

                return

            # Check it is going to be inserted

            # between any pair of Nodes (left, right)

            leftNode = self.headNode

            rightNode = self.headNode.nextNode

            while rightNode != None:
```

```
            if newNode < rightNode:
                leftNode.nextNode = newNode
                newNode.nextNode = rightNode
                return
            leftNode = rightNode
            rightNode = rightNode.nextNode
        # Once we reach here it must be added at the tail
        leftNode.nextNode = newNode
    else:
        # Check if it is going to be new head
        if newNode > self.headNode:
            self.__appendToHead(newNode)
            return
        # Check it is going to be inserted
        # between any pair of Nodes (left, right)
        leftNode = self.headNode
        rightNode = self.headNode.nextNode
        while rightNode != None:
            if newNode > rightNode:
                leftNode.nextNode = newNode
                newNode.nextNode = rightNode
                return
            leftNode = rightNode
            rightNode = rightNode.nextNode
        # Once we reach here it must be added at the tail
        leftNode.nextNode = newNode

# overloading operator
```

```python
def __str__(self):
    # We start at the head
    output =""
    node= self.headNode
    firstNode = True
    while node != None:
        if firstNode:
            output = node.__str__()
            firstNode = False
        else:
            output += (',' + node.__str__())
            node= node.nextNode
    return output

def __len__(self):
    return self.length

def __getitem__(self, key):
    node = self.headNode
    for i in range(key):
        node = node.nextNode
    return node
```

CLInterface Class written by Kaeden

```
from helpers import sort_expressions, Tokenizer

from Expression import Expression

from collection import SortedList


class CLInterface:

    def __init__(self):

        self.__current_selection = None


        # Prints Init str when app is started

        print("*" * 65)

        print("* ST107 DSAA: Expression Evaluator & Sorter                     *")

        print("*" + ("-" * 63) + "*")

        print("*  - Done by: Yeo Sheen Hern (1902257) & Kaedan Tan (1935529)   *")

        print("*  - Class DIT/FT/2B/11                                          *")

        print("*" * 65)


    # Prompts user for selection

    def selection_menu_prompt(self):

        self.__current_selection = None


        #built prompt

        prompt = "\nPlease select your choice ('1','2','3')"

        prompt += '\n   1. Evaluate expression'

        prompt += '\n   2. Sort expressions'

        prompt += '\n   3. Exit'

        prompt += '\nEnter choice: '
```

```python
    #input and check for 1 2 or 3
    while self.__current_selection not in ['1', '2' ,'3']:
        self.__current_selection = input(prompt)
        if self.__current_selection not in ['1', '2' ,'3']:
            print("Invalid input. Please input either '1', '2', or '3'\n")

# Evaluates Input Expression (Selection 1)
def evaluate_expression(self):
    exp_str = None
    # If invalid input, keep prompting for input
    while exp_str is None:
        exp_str = input('\nPlease enter the expression you want to evaluate:\n')
        try:
            exp_str = exp_str.replace(' ', '')
            expression = Expression(exp_str)
            expression.parse_tree()
        except Exception as e:
            exp_str = None
            print('Invalid Expression, ' + str(e))

    # checking for print order
    orderprint_selection = self.__print_order_selection()

    if orderprint_selection == "1":
        expression.print_preorder()
    elif orderprint_selection == "2":
        expression.print_inorder()
```

```
        elif orderprint_selection == "3":

            expression.print_postorder()


        # writes into a history file

        with open('./history.txt', 'a') as history_file:

            history_file.write("Expression {} evaluates to:
{:.3f}\n".format(str(expression), expression.val))

        with open('./input/input_history.txt', 'a') as history_file:

            history_file.write('\n' + str(expression))


        # printing

        print("\nExpression evaluates to:\n{:.3f}".format(expression.val))


    # Read Write File and Evaluate Expression (Selection 2)

    def sort_evaluate_expression(self):

        #input file prompt

        while True:

            # tries to read the file

            try:

                readfile = input("Please enter input file: ")

                with open('./input/'+readfile, 'r') as input_file:

                    input_file = input_file.read()

            except:

                print("The file does not exist. Please enter a valid input (e.g.
input.txt)")

                continue

            break
```

```
        #output file prompt
        while True:
            outfile = input("Please enter output file: ")
            # checks if filename has .txt
            if outfile[-4:] != '.txt':
                print("Not a valid filename. Please enter a valid filename ending with .txt (e.g. output.txt)")
                continue

            # tries to create/open the file
            try:
                output_file = open('./output/'+outfile, 'w')
            except:
                print('Not a valid filename. Please enter a valid filename (e.g. output.txt)\nA valid filename cannot include * . " / \ [ ] : ; | ,')
                continue
            break

        # splits inputfiles by \n new lines, if file has no line breaks it is considered as one expression

        input_file = input_file.splitlines()

        # sorting prompt
        sort_method = self.__print_sort_selection()
        ascending_check = self.__print_asc_check()
        sort_by = self.__print_sort_by_selection()

        sort_by_value = None
```

```
        if sort_by == "2":
            while True:
                sort_by_value = input('\nPlease enter the value (operator/number) you want to sort by:\n')
                try:
                    sort_by_value = sort_by_value.replace(' ', '')
                    if sort_by_value == "":
                        raise Exception("Please input a value")
                    tokenizer = Tokenizer(sort_by_value)
                    sort_by_value = list(tokenizer.generate_tokens())
                    if len(sort_by_value) > 1:
                        raise Exception("Value cannot be an expression")
                except Exception as e:
                    sort_by_value = None
                    print('Invalid input, ' + str(e))
                    continue
                break
            sort_by_value = sort_by_value[0]

        print("\n\n>>>Evaluation and sorting started:\n")

        # parses and evauates the input_file after splitting to an array by line breaks
        exp_list = self.__parsefile(input_file, sort_method, ascending_check, sort_by_value)

        # defining variables for printing
        current_val = None
        print_str = ""
```

```
        # building of final print and output write string
        if sort_by_value == None:
            for i in range(len(exp_list)):
                # if the current value not yet been printined print
                if current_val != exp_list[i].val:
                    print_str+= "\n\n*** Expressions with value=
{:.3f}\n".format(exp_list[i].val)
                    current_val = exp_list[i].val
                # always prints expression=>value
                print_str += ("{}==>{:.3f}\n".format(str(exp_list[i]),
exp_list[i].val))
        else:
            for i in range(len(exp_list)):
                # if the current value not yet been printined print
                if current_val != exp_list[i].sort_value_count():
                    print_str+= "\n\n*** Expressions with a total of {}x
'{}'s\n".format(exp_list[i].sort_value_count(), sort_by_value)
                    current_val = exp_list[i].sort_value_count()
                # always prints expression=>value
                print_str += ("{}==>{:.3f}\n".format(str(exp_list[i]),
exp_list[i].val))


        # prints actual output
        print(print_str)


        # writing output to file
        output_file.write(print_str)
        output_file.close()
```

```python
        print("\n\n>>>Evaluation and sorting completed!")



    def get_current_selection(self):
        return self.__current_selection


    # Private
functions_____________________________________________________________________________
_______________________________________________


    # Parses input file to expressions
    def __parsefile(self, input_file, sort_method, ascending_check, sort_by_value):
        # checks for sorting method and creates object accordingly
        if sort_method == "1":
            exp_list = SortedList(ascending_check)
        elif sort_method == "2":
            exp_list = []


        for i in range(len(input_file)):
            try:
                input_file[i].replace(' ', '')
                expression = Expression(input_file[i])
                expression.parse_tree()
                expression.set_sort_value(sort_by_value)
            except Exception as e:
                print("\nInvalid expression at line "+str(i+1)+". Skipping
expression")
                print(e)
```

```
            continue
        if sort_method =="1":
            exp_list.insert(expression)
        elif sort_method =="2":
            exp_list.append(expression)

    if sort_method == "2":
       sort_expressions(exp_list, ascending_check)
    return exp_list

# Prompts for
def __print_sort_by_selection(self):
    sort_by_selection = None

    #built prompt
    prompt = "\nPlease select your choice ('1','2')"
    prompt += '\n   1. Sort by value then length'
    prompt += '\n   2. Sort by count of user input'
    prompt += '\nEnter choice: '

    #input and check for 1 2 or 3
    while sort_by_selection not in ['1', '2']:
        sort_by_selection = input(prompt)
        if sort_by_selection not in ['1', '2']:
            print("Invalid input. Please input either '1' or '2'\n")

    return sort_by_selection
```

```python
# Prompts for print order selection
def __print_order_selection(self):
    orderprint_selection = None

    #built prompt
    prompt = "\nPlease select your choice ('1','2','3')"
    prompt += '\n   1. Print Preorder'
    prompt += '\n   2. Print Inorder'
    prompt += '\n   3. Print Postorder'
    prompt += '\nEnter choice: '

    #input and check for 1 2 or 3
    while orderprint_selection not in ['1', '2' ,'3']:
        orderprint_selection = input(prompt)
        if orderprint_selection not in ['1', '2' ,'3']:
            print("Invalid input. Please input either '1', '2', or '3'\n")

    return orderprint_selection

# prompts sort
def __print_sort_selection(self):
    #built prompt
    prompt = "\nPlease select your choice ('1','2')"
    prompt += '\n   1. Sorted list'
    prompt += '\n   2. Merge sort'
    prompt += '\nEnter choice: '

    sort_method = None
```

```
    #input and check for 1, 2
    while sort_method not in ['1', '2']:
        sort_method = input(prompt)
        if sort_method not in ['1', '2']:
            print("Invalid input. Please input either '1' or '2'\n")
    return sort_method

#prompts sort type
def __print_asc_check(self):
    #built prompt
    prompt = "\nPlease select your choice ('1','2')"
    prompt += '\n   1. Sort ascending'
    prompt += '\n   2. Sort descending'
    prompt += '\nEnter choice: '

    ascending_check = None

    #input and check for 1, 2
    while ascending_check not in ['1', '2']:
        ascending_check = input(prompt)
        if ascending_check not in ['1', '2']:
            print("Invalid input. Please input either '1' or '2'\n")
    return ascending_check
```

Sort_expressions Function written by Kaeden

```
#Can add more sorting algorithms while using sort_expressions as control function


def sort_expressions(input_list, ascending_check):

    return mergeSort(input_list, ascending_check)


def mergeSort(input_list, ascending_check):

    if len(input_list) > 1:

        mid = int(len(input_list)/2)


        leftHalf = input_list[:mid]

        rightHalf = input_list[mid:]


        mergeSort(leftHalf, ascending_check)

        mergeSort(rightHalf, ascending_check)


        leftIndex,rightIndex,mergeIndex = 0,0,0

        mergeList = input_list


        #checking for ascend descend

        if ascending_check == '1':

            #sorts ascending

            while leftIndex < len(leftHalf) and rightIndex < len(rightHalf):

                if leftHalf[leftIndex] < rightHalf[rightIndex]:

                    mergeList[mergeIndex] = leftHalf[leftIndex]

                    leftIndex+=1

                else:
```

```
                mergeList[mergeIndex] = rightHalf[rightIndex]

                rightIndex+=1

            mergeIndex+=1

    else:

        #sorts descending

        while leftIndex < len(leftHalf) and rightIndex < len(rightHalf):

            if leftHalf[leftIndex] < rightHalf[rightIndex]:

                mergeList[mergeIndex] = rightHalf[rightIndex]

                rightIndex+=1

            else:

                mergeList[mergeIndex] = leftHalf[leftIndex]

                leftIndex+=1

            mergeIndex+=1

    # Handle those items still left in the left Half

    while leftIndex < len(leftHalf):

        mergeList[mergeIndex] = leftHalf[leftIndex]

        leftIndex+=1

        mergeIndex+=1

    # Handle those items still left in the right Half

    while rightIndex < len(rightHalf):

        mergeList[mergeIndex] = rightHalf[rightIndex]

        rightIndex+=1

        mergeIndex+=1
```