CS211 - Data Structures and Algorithms

Lab 07

Instructor: Dr. Sharaf Hussain
E-mail: shhussain@uit.edu

Semester: Fall, 2021

1 Objective

The purpose of this lab session is to implement **Linked List** data structure and its applications.

2 Instructions

You have to perform the following tasks yourselves. Raise your hand if you face any difficulty in understanding and solving these tasks. **Plagiarism** is an abhorrent practice and you should not engage in it.

3 How to Submit

- Submit lab work in a single .py file on Google Classroom. (No other format will be accepted)
- Lab work file name should be saved with your roll number (e.g. 19a-001-SE_LW04.py)
- Submit home work in a single .py file on Google Classroom. (No other format will be accepted)
- Lab work file name should be saved with your roll number (e.g. 19a-001-SE_HW04.py)

Task 1 - Sparse Matrix

Implement following **Sparse Matrix** using linked list data structure. Implement subtract, multiply, and transpose methods in the code.

```
# Implementation of the Sparse Matrix ADT using an array of linked lists
.
from Chapter2.myarray import Array

class SparseMatrix :
  # Creates a sparse matrix of size numRows x numCols initialized to 0.
  def __init__( self, numRows, numCols ):
    self._numCols = numCols
    self._listOfRows = Array( numRows )

  # Returns the number of rows in the matrix.
  def numRows( self ):
    return len( self._listOfRows )

  # Returns the number of columns in the matrix.
  def numCols( self ):
    return self._numCols

  # Returns the value of element (i,j): x[i,j]
  def __getitem__( self, ndxTuple ):
    row = ndxTuple[0]
    col = ndxTuple[1]
```

```
        predNode = None
        curNode = self._listOfRows[row]
        while curNode is not None and curNode.col != col:
          predNode = curNode
          curNode = curNode.next
        if curNode is not None and curNode.col == col:
          return curNode.value

      # Sets the value of element (i,j) to the value s: x[i,j] = s
      def __setitem__( self, ndxTuple, value ):
        row = ndxTuple[0]
        col = ndxTuple[1]
        predNode = None
        curNode = self._listOfRows[row]
        while curNode is not None and curNode.col != col :
          predNode = curNode
          curNode = curNode.next

        # See if the element is in the list.
        if curNode is not None and curNode.col == col :
          if value == 0.0 : # remove the node.
            if curNode == self._listOfRows[row] :
              self._listOfRows[row] = curNode.next
            else :
              predNode.next = curNode.next
          else : # change the node's value.
            curNode.value = value

        # Otherwise, the element is not in the list.
        elif value != 0.0 :
          newNode = _MatrixElementNode( col, value )
          newNode.next == curNode
          if curNode == self._listOfRows[row] :
            self._listOfRows[row] = newNode
          else :
            predNode.next = newNode

      # Scales the matrix by the given scalar.
      def scaleBy( self, scalar ):
        for row in range( self.numRows() ) :
          curNode = self._listOfRows[row]
          while curNode is not None :
            curNode.value *= scalar
            curNode = curNode.next

      # Creates and returns a new matrix that is the transpose of this
    matrix.
      def transpose( self ):
        ...

      # Matrix addition: newMatrix = self + rhsMatrix.
      def __add__( self, rhsMatrix) :
        # Make sure the two matrices have the correct size.
        assert rhsMatrix.numRows() == self.numRows() and \
        rhsMatrix.numCols() == self.numCols(), \
        "Matrix sizes not compatable for adding."

        # Create a new sparse matrix of the same size.
        newMatrix = SparseMatrix( self.numRows(), self.numCols() )

        # Add the elements of this matrix to the new matrix.
        for row in range( self.numRows() ) :
          curNode = self._listOfRows[row]
          while curNode is not None :
            newMatrix[row, curNode.col] = curNode.value
            curNode = curNode.next

        # Add the elements of the rhsMatrix to the new matrix.
        for row in range(rhsMatrix.numRows() ) :
          curNode = rhsMatrix._listOfRows[row]
          while curNode is not None :
            value = newMatrix[row, curNode.col]
            value += curNode.value
```

```
            newMatrix[row, curNode.col] = value
            curNode = curNode.next

        # Return the new matrix.
        return newMatrix

    # --- Matrix subtraction and multiplication ---
    def __sub__( self, rhsMatrix ) :
        ...
    # def __mul__( self, rhsMatrix ) :
        ...

# Storage class for creating matrix element nodes.
class _MatrixElementNode :
def __init__( self, col, value ) :
self.col = col
self.value = value
self.next = None
```

Task 2 - Polynomial Equation

Implement following **Polynomial Equation** using linked list data structure. Implement subtract, multiply, and evaluation methods in the code.

```
#Implementation of the Polynomial ADT using a sorted linked list
class Polynomial:
    def __init__(self, degree = None, coefficient = None):
        if degree is None:
            self._polyHead = None
        else:
            self._polyHead = _PolyTermNode(degree, coefficient)
        self._polyTail = self._polyHead

    def degree(self):
        if self._polyHead is None:
            return -1
        else:
            return self._polyHead.degree

    def __getitem__(self, degree):
        assert self.degree() >= 0, \
            "Operation not permitted in empty polynomial."
        curNode = self._polyHead
        while curNoe is not None and curNode.degree >= degree:
            curNode = curNode.next
        if curNode is None or curNode.degree != degree:
            return 0.0
        else:
            return curNode.degree

    def evaluate(self, scalar):
        pass

    def __add__(self, rhsPoly):
        assert self.degree() >= 0 and rhsPoly.degree() >= 0, \
            "Addition only allowed in non-empty Polynomials"
        newPoly = Polynomial()
        nodeA = self._polyHead
        nodeB = self._polyHead

        while nodeA is not None and nodeB is not None:
            if nodeA.degree > nodeB.degree:
                degree = nodeA.degree
                value = nodeA.coefficient
                nodeA = nodeA.next
            elif nodeA.degree < nodeB.degree:
                degree = nodeB.degree
                value = nodeB.coefficient
                nodeB = nodeB.next
```

```
            else:
                degree = nodeA.degree
                value = nodeA.coefficient + nodeB.coefficient
                #adds when degree is same
                nodeA = nodeA.next
                nodeB = nodeB.next
            newPoly._appendTerm(degree, value)

        while nodeA is not None:
            newPoly._appendTerm(nodeA.degree, nodeA.coefficient)
            nodeA = nodeA.next
            #if the list itself is longer
        while nodeB is not None:
            newPoly._appendTerm(nodeB.degree, nodeB.coefficient)
            nodeB = nodeB.next
        return newPoly


    def __sub__(self, rhsPoly):
        pass

    def __mul__(self, rhsPoly):
        pass

    #Helper method for appending terms in the polynomial
    def _appendTerm(self, degree, coefficient):
        if coefficient != 0.0:
            newTerm = _PolyTermNode(degree, coefficient)
            if self._polyHead is None:
                self._polyHead = newTerm
            else:
                self._polyTail.next = newTerm
            self._polyTail = newTerm


class _PolyTermNode(object):
    def __init__(self, degree, coefficient):
        self.degree = degree
        self.coefficient = coefficient
        self.next = None
```