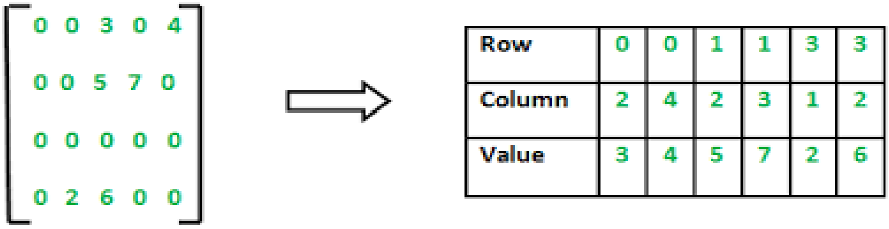
Programming Assignment 11

Sparse Matrix

Definition: A sparse matrix is a matrix in which very few elements are non-zero.

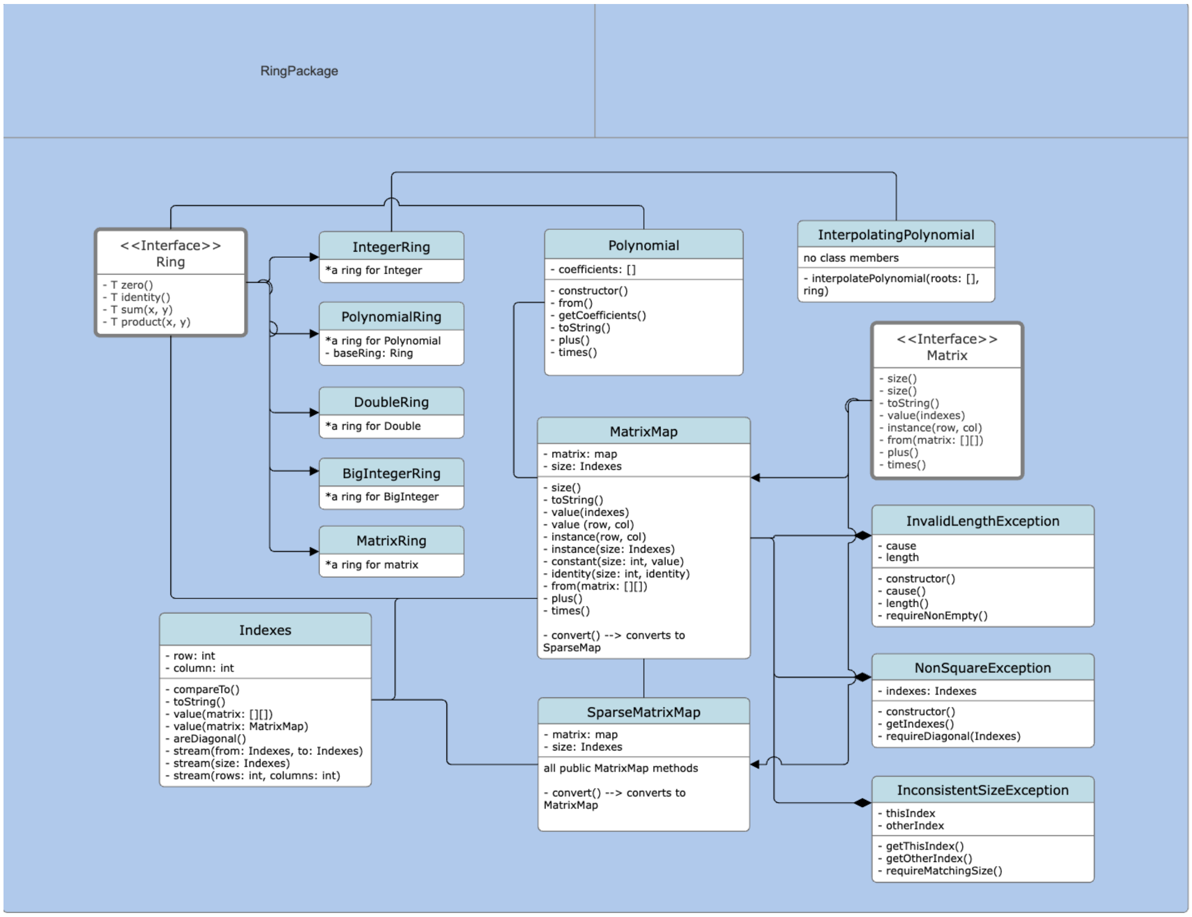


$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Sparse matrices can be expressed as separate tables row, column, and value entries. The simplest java implementation for this table is likely a two-dimensional array which begins indexing at [1][1].

Ring Package



Class/Interface Captured Abstractions:

Ring<T>:

- an interface describing the operations of a ring
- sits on top of the Ring hierarchy
- contains only abstract methods

BigIntegerRing, DoubleRing, IntegerRing:

- define Ring operations for their respective data type
- implements Ring interface
- constructor/public methods
  - identity()
  - zero()
  - sum(x, y)
  - product(x, y)

Rings:

- provides Ring powered functionality to reduce a list of arguments to one result either through addition or multiplication
- external class in the hierarchy
- constructor/
- methods
  - reduce(List<T> args, T zero, BinaryOperator<T> accumulator)
  - sum(List<T> args, Ring<T> ring)
  - product(List<T> args, Ring<T> ring)

Polynomial<T>:

- defines a Polynomial object which is comprised of a list of coefficients, meant to represent the polynomial in factored form
- external class in the hierarchy
- constructor/methods
  - Polynomial(List<T> coefficients)
  - from(List<S> coefficients)
  - List<T> getCoefficients()
  - toString()
  - plus(Polynomial<T> other, Ring<T> ring)

    - times(Polynomial<T> other, Ring<T> ring)
- private data structures:
    - List<T> coefficients

PolynomialRing<T>:

- defines Ring operations on Polynomials
- implements Ring interface
- contains
    - Ring<T> baseRing → containment, aggregation
- constructor/public methods
    - PolynomialRing(Ring<T> ring)
    - polynomialRing<T> instance(Ring<T> ring)
    - zero()
    - identity()
    - sum(Polynomial<T> x, Polynomial<T> y)
    - product(Polynomial<T> x, Polynomial<T> y)

InterpolatingPolynomial:

- a class representing the polynomial interpolated through a set of roots
- external class in hierarchy
- constructor/public methods
    - interpolatePolynomial(List<Integer> rootList, Ring<Integer> ring)

Indexes:

- a record representing the coordinates of a matrix
- external class in the hierarchy
- constructor/public methods
    - compareTo(Indexes o)
    - toString()
    - value(S[][] matrix)
    - value(MatrixMap<S> matrix)
    - areDiagonal()
    - stream(Indexes from, Indexes to)
    - stream(Indexes size)

   - stream(int rows, int columns)

Matrix<T>:

- an interface for the matrix classes
- abstract methods:
   - size()
   - value(Indexes indexes)
   - instance(int rows, int columns, Function<Indexes, S>)
   - constant(int size, S value)
   - identity(int size, S zero, S identity)
   - plus(MatrixMap<T> other, BinaryOperator<T> plus)
   - times(MatrixMap<T> other, Ring<T> ring)

MatrixMap<T>:

- defines a matrix with which certain operations can be performed
- implements Matrix
- contains
   - Map<Indexes, T> matrix → containment, aggregation
   - Indexes size → containment, aggregation
   - class InvalidLengthException, containment, aggregation
   - class InconsistentSizeException, containment, aggregation
   - class NonSquareException, containment, aggregation
- constructor/public methods
   - MatrixMap(Map<Indexes, T> matrix)
   - Indexes size()
   - String toString()
   - value(Indexes indexes)
   - value(int row, int column)
   - instance(int rows, int columns, Function<Indexes, S> valueMapper)
   - instance(Indexes size, Function<Indexes, S> valueMapper)
   - constant(int size, S value)
   - identity(int size, S zero, S identity)
   - from(S[][] matrix)
   - plus(MatrixMap<T> other, BinaryOperator<T> plus)
   - times(MatrixMap<T> other, Ring<T> ring)
   - convertToSparse(Ring<T> ring)
- private data structures

    - Map<Indexes, T> matrix

SparseMatrixMap<T>:

- defines a sparse matrix, which the user has a choice to instantiate along with a normal matrix
- external class in the hierarchy, extends Matrix interface
- constructor/public methods
    - constructor
    - size()
    - value(Indexes indexes)
    - instance(int rows, int columns, Function<Indexes, S> valueMapper)
    - instance(Indexes size, Ring<T> ring, Function<Indexes, S> valueMapper)
    - constant(int size, Ring<T> ring, S value)
    - identity(int size, Ring<T> ring)
    - plus(Matrix<T> other, Ring<T> ring, BinaryOperator<T> plus)
    - times(Matrix<T> other, Ring<T> ring)
    - convertToStandard(Ring<T> ring)
- private data structures
    - Map<Indexes, T> matrix

MatrixRing<T>:

- defines a Ring for the matrix classes
- extends Ring
- contains
    - Ring<T> baseRing
- constructor/public methods
    - MatrixRing(PolynomialRing<T> ring)
    - MatrixRing<T> instance(PolynomialRing<T> ring)
    - sum(Matrix<T> x, Matrix<T> y)
    - product(Matrix<T> x, Matrix<T> y)

Pseudocode for Complex Methods:

```
class Rings {

     reduce(list of args, zero, accumulator) {
        foundAny ← false;
        result ← zero;
        for each element in args do
            if (not foundAny) {
                foundAny ←true;
                result ←element;
            } else {
                result ←accumulator(result, element);
            }
        }
        return result;
    }
}

class Polynomial {

    plus(other Polynomial, ring) {

        a ← list of current coefficients
        b ← list of other coefficients
        maxLength ← max(a, b)
        sum_list ← new list of size maxLength

        aIter ← iterator for a
        bIter ← iterator for b

        while (aIter has next or bIter has next) {
            a_addend ← addend from a
            b_addend ← addend from b

            sum ← ring.sum(a_addend, b_addend)
            add sum to sum_list
        }
        return Polynomial(sum_list)
    }

    times(other Polynomial, ring) {

        a ← list of current coefficients
        b ← list of other coefficients

        productLength ← computeProductLength(a, b)
        product_list ← new list of size productLength

        a_start ← 0
        for i ← 0 until i < productLength {

            product ← ring.zero()
            a_start ← computeStartIndex(i, a_start, b)
            aIter ← iterator for a
            bIter ← iterator for b

            while (aIter has next and bIter has previous) {
                a_factor ← next of aIter
                b_factor ← previous of bIter
                result ← ring.product(a_factor, b_factor)
                product = ring.sum(product, result)
            }
            add product to product_list
        }
        return Polynomial(product_list)
    }
}

class InterpolatingPolynomial {

    interpolatePolynomial(list of roots)
        if list of roots is null
            throw an appropriate exception
        require elements in roots are not null
        roots ← param list of zeros
        x ← Multiplicative identity of a polynomial
        factors ← empty list of factors
        for each root in roots
            create new Polynomial ← (x, root * -1)
            insert polynomial → factors
        return product(factors, ring)
}

class MatrixMap implements Matrix {

    instance(rows, columns, valueMapping function) {
        matrix ← new Map
        indexes ← new list of indexes(rows, columns)
        populate(valueMapper, matrix, indexes)
        return a new MatrixMap(matrix)
    }

    populate(Function valueMapper, Map matrix, List indexes) {
        for each index in indexes {
            value ← valueMapper.apply(index)
            matrix.put(index, value)
        }
    }

    times(other MatrixMap, ring) {
        list of products ← new list
        length ← size of current matrix rows
        return instance(size, (index) → {
            list products ← new list
            for i ← 0 until i < length {
                products.add(product of (value of row(), i) and
                other Matrix value of i, column())
            }
            return Rings.sum(products)
        }
    }

    SparseMatrixMap convertToSparse() {
        return instance(size of this matrix, at each index, value
at this index -> index of sparse)
    }
}

class SparseMatrixMap implements Matrix {

    Map matrix
    int size

    instance(rows, columns, ring valueMapping function) {
        check if params are null
        check if rows and columns are valid
        matrix <- new map
        create a list of indexes with the given size from params
        for each index in indexes
            value <- apply valueMapper
            if value is not ring.zero
                put the mapping into the map
        return a new SparseMatrix(copy of matrix, ring)
    }
}
```

Error Handling Decisions

I believe the best error handling method for this implementation would be to use exceptions in a localized fashion. This can be paired with supplemental information to aid error tracking and further organize failure capture information. For the rest of the package, the potential for errors is significantly lower, meaning errors can be dealt with on a case by case basis in a localized fashion.

Testing and Stress Testing

Unit Testing:

1. test methods in MatrixMap primarily
   a. MatrixMap utilizes classes and methods from most other classes, which means many of them can be implicitly tested
2. test MatrixMap for various data types, which can test different types of Rings
3. test corner cases only after full code and branch coverage is complete
   a. this is to ensure the edges of the program are validated after nominal cases have been processed

Stress Tests:

There are many possible stress tests that are executable for the classes in Ring Package. My approach would be to select the classes which contain private data structure members and test the functionality of those classes by loading an extremely large amount of data into the data structures. One way to measure performance under these conditions is to monitor execution time for each method call.