COSC160: Data Structures Matrices

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Case Study: Matrices and Sparse Matrices

Case Study: Matrix Structure

- Design Matrix Class, to help facilitate basic mathematical matrix operations
  - Design Notes:
    - Matrix contains real numbers in each entry
    - Matrix Addition
    - Matrix Transpose
    - Matrix Multiplication

Matrices

- A matrix is a table with rows and columns.
  - EG: A 3 x 2 matrix: 3 rows and 2 columns
    - Matrix entries are generally indexed by a row, column tuple, : matrix(2,1) is 6.
      - using mathematical notation the indexing usually starts at 1

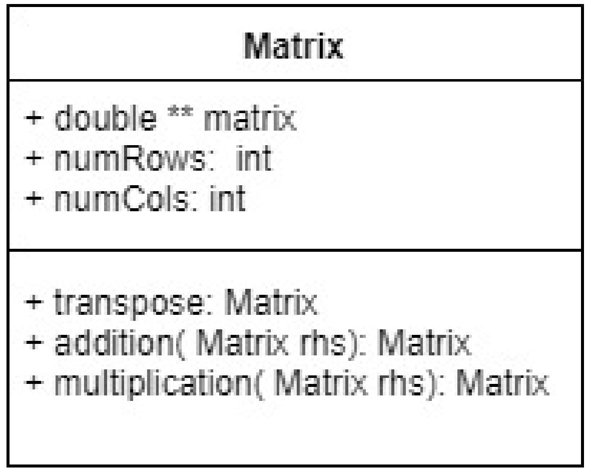
$$\begin{bmatrix} 1 & 5 \\ 6 & 2 \\ 9 & 7 \end{bmatrix}$$

- Matrix implementation
  - Design decision: Should the structure be represented as a 2-d array or 2-d chaining structure?

GEORGETOWN UNIVERSITY

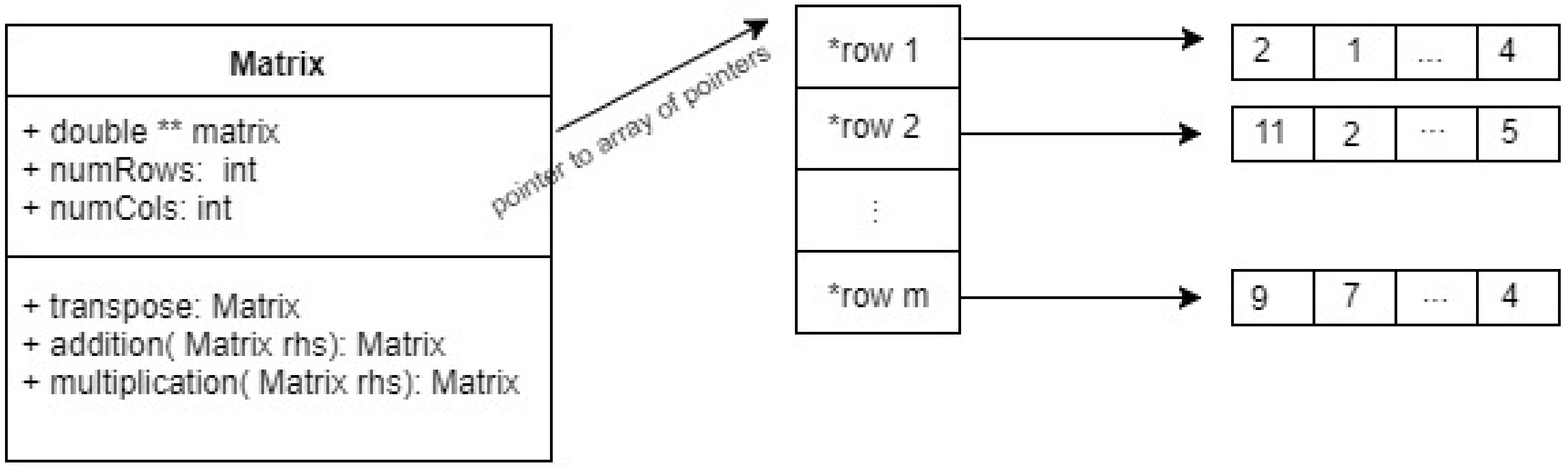
Matrix Implementation (UML)

- Intuitively, we can use a 2-d array to implement a matrix.
  - Thus we can access *ith* row and *jth* col: m[i-1][j-1].
  - We can design a Matrix class with
    - member variables
      - double ** matrix   // or use template
      - int numRows
      - int numCols
    - member methods
      - Matrix add(Matrix op);
      - Matrix transpose();
      - Matrix multiply(Matrix op);



Why is our member a pointer to a pointer?

- Matrix member: double ** matrix;
  - Size of matrix is variable. Thus our we can implement a matrix as a pointer to an array of pointers.
  - Many programming languages also allow you to simply allocate a multi-dimensional array with one instruction: matrix[numRows][numCols]
    - Question: Will this matrix need to be allocated on the heap?



Matrix Transpose

- Mathematically

$$\forall_{i,j}\ M^T_{i,j} = M_{j,i}$$

- Code Snippet

```
Algorithm 1
Require: keyword 'this' is used to refer to calling Matrix object
  function TRANSPOSE($this$)
    $cols \leftarrow this \rightarrow numRows$
    $rows \leftarrow this \rightarrow numCols$
    $trans \leftarrow$ initialize new Matrix rows x cols
    for $i \leftarrow 1$ to $rows$ do
      for $j \leftarrow 1$ to $cols$ do
        $trans \rightarrow matrix[i][j] \leftarrow this \rightarrow matrix[j][i]$
    return $trans$
```

- Analysis (also practical)
  - Time?
  - Space?
    - Note: thrashing may be unavoidable
  - Comments?
  - Can we improve?
    - With some pointer arithmetic, we could do an in-place swap as the correct number of locations are indeed allocated … but would we want to manipulate the calling object … ?

GEORGETOWN UNIVERSITY

Implementation Notes: row major vs. col major

- Some matrix operations can be fairly inefficient when the matrices become very large
  - Remember:
    - Memory / Caching issues can compound this issue (severely!)
    - Thrashing
- How might we mitigate these issues when implementing an add algorithm?
  - Traverse the matrices smartly

Matrix Add

- Mathematically

$$C = A + B \equiv \forall_{i,j}\ c_{i,j} = a_{i,j} + b_{i,j}$$

- Pseudocode

```
Algorithm 1
Require: keyword 'this' is used to refer to calling Matrix object
  function ADDITION(Matrix this, Matrix rhs)
    $rows \leftarrow this \rightarrow numRows$
    $cols \leftarrow this \rightarrow numCols$
    $result \leftarrow$ initialize new Matrix rows x cols
    for $i \leftarrow 1$ to $rows$ do
      for $j \leftarrow 1$ to $cols$ do
        $result \rightarrow matrix[i][j] \leftarrow (this \rightarrow matrix[i][j]) + (rhs \rightarrow matrix[i][j])$
    return $result$
```

- Analysis
  - Space (also practical):
    - Pass by reference or pass by copy? Heap or Stack Allocation?
    - If allocation on the heap, there must be an infrastructure in place to deallocate this matrix when necessary.
    - O(row x col)
  - Time:
    - O(row x col)

GEORGETOWN UNIVERSITY

Matrix Multiply

- Try this exercise at home: implement matrix multiply

Summary: Analysis of Matrix Example

- Implementation Decisions and Repercussions
  - Implemented as 2-d array
    - Provides for efficient storage and retrieval (assuming size won't change)
  - ADD Algorithm scans assuming row-major memory storage
    - Knowledge of underlying memory details allows for efficient traversal (not improvement in complexity analysis, but improvement in practical application)
    - Transpose?
      - May require swap of values at different ends of matrix which can cause caching delays.
  - If new matrices resulting from operations are allocated on the heap …
    - Presents deallocation issues and memory concerns. Probably best to use OOP practices.
    - Trade off
      - Reliability: keep the object itself "light weight" (use pointers for members as needed), and manage memory deallocation with the implicit call to the destructor (when the objects scope is destroyed)
      - Cost: potential unnecessary copy (likely by copy constructor)

- Time and Space analyses
  - Operations are efficient and on the order of the minimal number of operations necessary to complete the computations

GEORGETOWN UNIVERSITY

Case Study: Sparse Matrix Structure

- Lengthy processing times for large matrices are sometimes unavoidable
  - Each operation requires some minimum amount of computation
- In some scenarios, a matrix may have a “small” number of non-zero entries.
  - Such matrices are called sparse matrices.
    - Adding and multiplying with zero is trivial.
  - If we can find an efficient representation of a sparse matrix which does not require the traversal of zero-valued entries, we can improve the computational complexity.

Sparse Matrices

- Sparse Matrix
  - Using standard representation, we will need to use memory spaces on the order of numRows x numCols.
    - $\begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
  - Ignoring zero-valued entries we can significantly reduce the space requirements (and possibly time complexity for operations).
    - How might we represent a sparse matrix compactly?

Sparse Matrices

- We can represent all non-zero values as a 3-tuple: (row, col, val)
  - All other values are zero
- Example

$$\begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{ccc} \text{Row} & \text{Col} & \text{Val} \\ (0, & 0, & 9) \\ (1, & 5, & 2) \end{array}$$

- Analysis
  - Space
    - There is a clear reduction in space, which now depends on the number of non-zero values rather than number of rows and columns

Sparse Matrices

- Consider, when designing a data structure, *Form Follows Function*
- Investigate the various operations (functions) needed.
- Based on your investigation you can identify which design decisions will lead to more efficient implementations
  - Contiguous vs non-Contiguous allocation
    - May affect space and/or time complexity (of operations)
  - Assumptions or requirements concerning the VALID STATE(S) of the data structure
    - Imposing certain requirements may improve time and/or space complexity

Sparse Matrix: Addition

- Example: ***Sparse*** Matrix Addition
  - Assume both matrices are 3 x 5.

Sparse Matrix Representations

$$\begin{array}{lll}(0, & 2, & 9) \\ (2, & 3, & 2) \\ (2, & 4, & 1)\end{array} + \begin{array}{lll}(0, & 1, & 3) \\ (2, & 4, & 2)\end{array} = ?$$

Actual Matrices

$$\begin{bmatrix}0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0\end{bmatrix} + \begin{bmatrix}0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0\end{bmatrix}$$

Lets investigate an add method. Given the format of our matrix,

1) what is the process to accomplish addition?
2) how can we make this efficient?

Sparse Matrix Addition (version 1)

- Goal: c = a + b
- Pseudocode Idea:
  - For each row in a, check all rows in b for a match.
    - Case 1: if b has a term at same index,
      - then create a new term, add a and b val and append to c.
    - Case 2: b does not have a matching term
      - Append the a-term to c
  - For each row in b, check all rows in a for match.
    - Case 1: if a has a term at same index,
      - Do nothing! Term should already be added.
    - Case 2: b does not have a matching term
      - Append the b-term to c

$$\begin{matrix}(0, & 2, & 9)\\(2, & 3, & 2)\\(2, & 4, & 1)\end{matrix} + \begin{matrix}(0, & 1, & 3)\\(2, & 4, & 2)\end{matrix} = \begin{matrix}(0, & 2, & 9)\\(2, & 3, & 2)\\(2 & 4, & 3)\\(0, & 1, & 3)\end{matrix}$$

- Time complexity analysis?
  - Embedded looping structure.
  - O(numTermsA x numTermsB)
  - Can we do better?

ALSO NOTE: If we choose array implementation, we will need to allocate c before we begin the appending. How can we get this accomplished?

Sparse Matrix Addition (version 2)

- Goal: c = a + b
- Improvement: If we require the tuples to be ordered, we can simplify the traversal scheme and reduce computation .
- Since the matrices are sorted by rows (and then cols), the result is a “row major” order, and thus we can sequentially scan each matrix simultaneously (illustrated with arrows). No embedded iteration! There will be 3 cases per iteration.
  - Case 1: The current term in a is earlier in the matrix than the current b term.
    - Result: append the a-term to c
  - Case 2: The current term in b is earlier in the matrix than the current a term.
    - Result: append the b-term to c
  - Case 3: The current terms for a and b have the same index.
    - Result: Add the values of a and b and append the new term to c

$$\begin{array}{l}\rightarrow (0,\quad 2,\quad 9)\\ \rightarrow (2,\quad 3,\quad 2)\\ \rightarrow (2,\quad 4,\quad 1)\end{array} + \begin{array}{l}\rightarrow (0,\quad 1,\quad 3)\\ \rightarrow (2,\quad 4,\quad 2)\end{array} = \begin{array}{l}(0,\quad 1,\quad 3)\\ (0,\quad 2,\quad 9)\\ (2\quad 3,\quad 2)\\ (2,\quad 4,\quad 3)\end{array}$$

- Time Complexity?
  - Traverse each structure only once!
  - O(numTermsA + numTermsB)
  - How does this compare to non-sparse matrix implementation?
  - When does it become “useful” to use sparse implementation as opposed to standard implementation?

Again note : If we choose array implementation, we will need to allocate c before we begin the appending. How can we get this accomplished?

Sparse Matrix Data Structure

- Investigation of our matrix representation

- ***Maintaining a valid state of a data structure***
  - Many matrix operations depend on the scanning of rows and columns, maintaining our structure such that it is *ordered by row scan* (or col) will improve efficiency of operations.
  - This **State** will need to be maintained (and possibly checked) for all operations.

  - State of Data Structure (design decision):
    - A list of 3-tuples,
    - As we have seen, it is beneficial to keep the terms in order (order by row-scan)
    - Design decision: array or chain … lets investigate.
      - Form Follows Function

Sparse Matrices

- Sparse Matrix **Transpose** Example
  - Solution simple right?
    - Swap the rows and columns in each triple -- NO
  - Concern
    - We must maintain a valid state of our structure (the order by row major scan)

Row	Col	Val
(1,	1,	9)
(2,	2,	2)
(2,	5,	1)
(3,	2,	3)
(4,	6,	5)
(4,	1,	8)

$\xrightarrow{\cancel{\text{transpose}}}$

Row	Col	Val
(1,	1,	9)
(2,	2,	2)
(5,	2,	1)
(2,	3,	3)
(6,	4,	5)
(1,	4,	8)

Observe: Requiring order of tuples complicates the transpose operation. If total row scan order is maintained, then transpose can still be performed in linear time, with a minor increase of computational steps to maintain order.

GEORGETOWN UNIVERSITY

Sparse Matrix Transpose (version 1)

- Goal b = a^T
- Pseudocode idea:
  - For each column c in a
    - For each term in a
      - Identify each term in column c, swap row and col values, and add term to b.
      - Note: order should be maintained using standard sequential scan.

a	b
(1, 1, 9)	(1, 1, 9)
(2, 2, 2)	(1, 4, 8)
(2, 5, 1)	(2, 2, 2)
(3, 2, 3)	(2, 3, 3)
(4, 6, 5)	(5, 2, 1)
(4, 1, 8)	(6, 4, 5)

- Time Complexity
  - O(numTerms * numColumns)
  - Can we do better?

GEORGETOWN UNIVERSITY

Sparse Matrix Transpose (version 2)

- Goal b = a^T
- Pseudocode idea (build jump table):
  - For each term in a
    - Count number of terms in each col.
  - For each col c in a
    - Construct jump table: Determine the start index for each row given count
  - For each term in a
    - Swap row and col values and add term to be using correct order determined by jump table
- Jump Table requires direct access
  - Array implementation
- Time Complexity
  - O(numTerms + numColumns)
  - FAST!

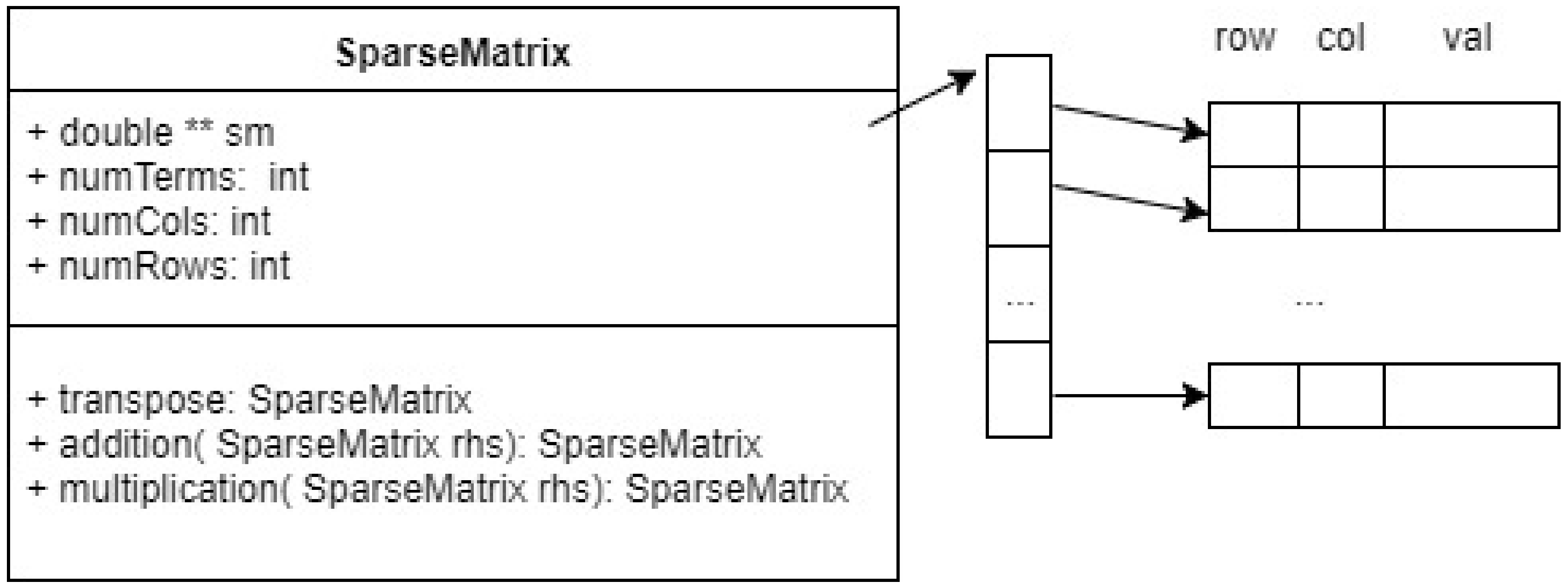
a	b
(1, 1, 9)	(1, 1, 9)
(2, 2, 2)	(1, 4, 8)
(2, 5, 1)	(2, 2, 2)
(3, 2, 3)	(2, 3, 3)
(4, 6, 5)	(5, 2, 1)
(4, 1, 8)	(6, 4, 5)

Observations

- Design Observations and Decisions.
  - Ordering tuples
    - Ordering tuples will reduce addition time complexity from quadratic to linear.
    - This is worth the increase in time complexity for the transpose operation.
    - Multiply?
    - Decision: Yes to order!
  - Chaining vs array
    - Addition. Size of resulting sparse matrix is unknown until addition is performed. Thus array maybe twice as slow.
    - Decision: Yes to chain!

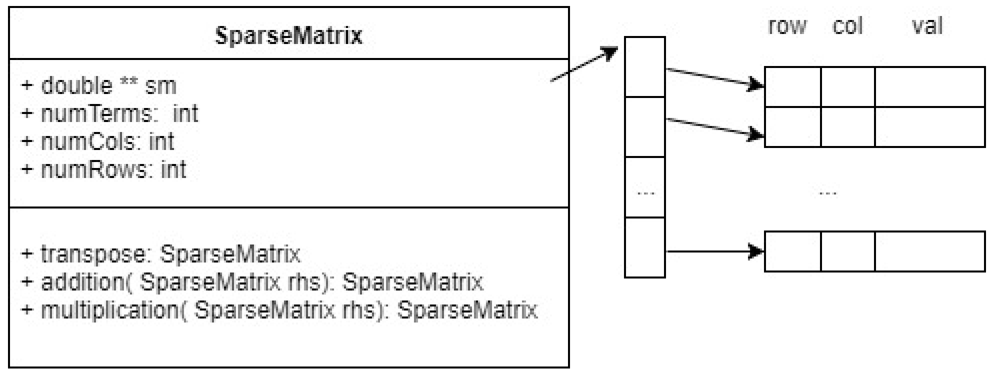
- For the sake of example, lets fully investigate chain vs array.

Sparse Matrices: Array Implementation


SparseMatrix
+ double ** sm
+ numTerms: int
+ numCols: int
+ numRows: int
+ transpose: SparseMatrix
+ addition(SparseMatrix rhs): SparseMatrix
+ multiplication(SparseMatrix rhs): SparseMatrix
row
col
val
...
...

Array Implementation: Transpose

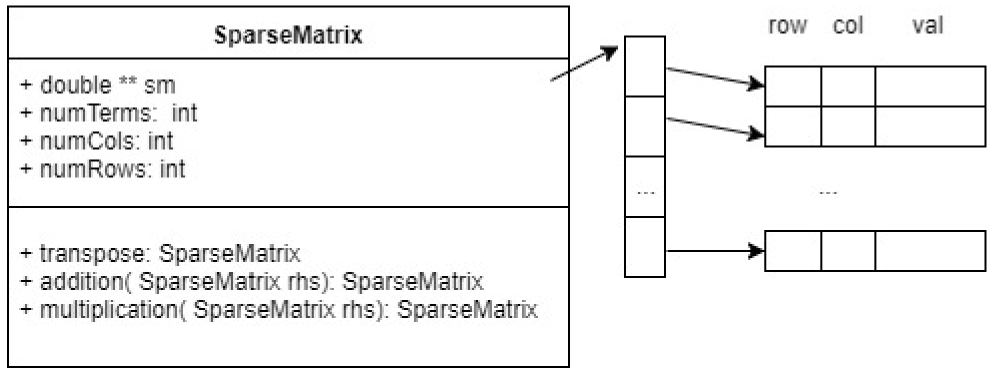
- Sparse Matrix Transpose (version 1)


SparseMatrix
+ double ** sm
+ numTerms: int
+ numCols: int
+ numRows: int
+ transpose: SparseMatrix
+ addition(SparseMatrix rhs): SparseMatrix
+ multiplication(SparseMatrix rhs): SparseMatrix
row
col
val
...
...

```
Algorithm 1
Require: keyword 'this' is used to refer to calling SparseMatrix object
  function TRANSPOSE(SparseMatrix this)
      $numTerms \leftarrow this \rightarrow numTerms$
      $result \leftarrow$ initialize new SparseMatrix numTerms by 3
      $index \leftarrow 1$
      for $i \leftarrow 1$ to $this \rightarrow numCols$ do
          for $j \leftarrow 1$ to $numTerms$ do
              if $(Xthis \rightarrow sm[j][1]) == i$ then
                  $result \rightarrow sm[index][0] \leftarrow this \rightarrow sm[j][1]$
                  $result \rightarrow sm[index][1] \leftarrow this \rightarrow sm[j][0]$
                  $result \rightarrow sm[index++][2] \leftarrow this \rightarrow sm[j][2]$
      return $result$
```

Array Implementation: Transpose

- Sparse Matrix Transpose (version 2)


SparseMatrix
+ double ** sm
+ numTerms: int
+ numCols: int
+ numRows: int
+ transpose: SparseMatrix
+ addition(SparseMatrix rhs): SparseMatrix
+ multiplication(SparseMatrix rhs): SparseMatrix
row
col
val
...
...

```
Algorithm 1
Require: keyword 'this' is used to refer to calling SparseMatrix object
  function TRANSPOSE(SparseMatrix this)
    $numTerms \leftarrow this \rightarrow numTerms$
    $result \leftarrow$ initialize new SparseMatrix numTerms
    $rowSize \leftarrow$ initialize new int array of length this $\rightarrow$ numCols
    $rowStart \leftarrow$ initialize new int array of length this $\rightarrow$ numCols
    for $i \leftarrow 0$ to $this \rightarrow numTerms - 1$ do
      $rowSize[this \rightarrow sm[i][1]]++$ // count num terms in each col
    for $i \leftarrow 0$ to $this \rightarrow numCols - 1$ do
      $rowStart[i] \leftarrow rowStart[i-1] + rowSize[i-1]$ // determine index of rows
    for $i \leftarrow 0$ to $numTerms - 1$ do
      $j \leftarrow rowStart[this \rightarrow sm[i,1]]$
      $result \rightarrow sm[j][0] \leftarrow this \rightarrow sm[i][1]$
      $result \rightarrow sm[j][1] \leftarrow this \rightarrow sm[i][0]$
      $result \rightarrow sm[j][2] \leftarrow this \rightarrow sm[i][2]$
      $rowStart[this \rightarrow sm[i,1]]++$
    return $result$
```

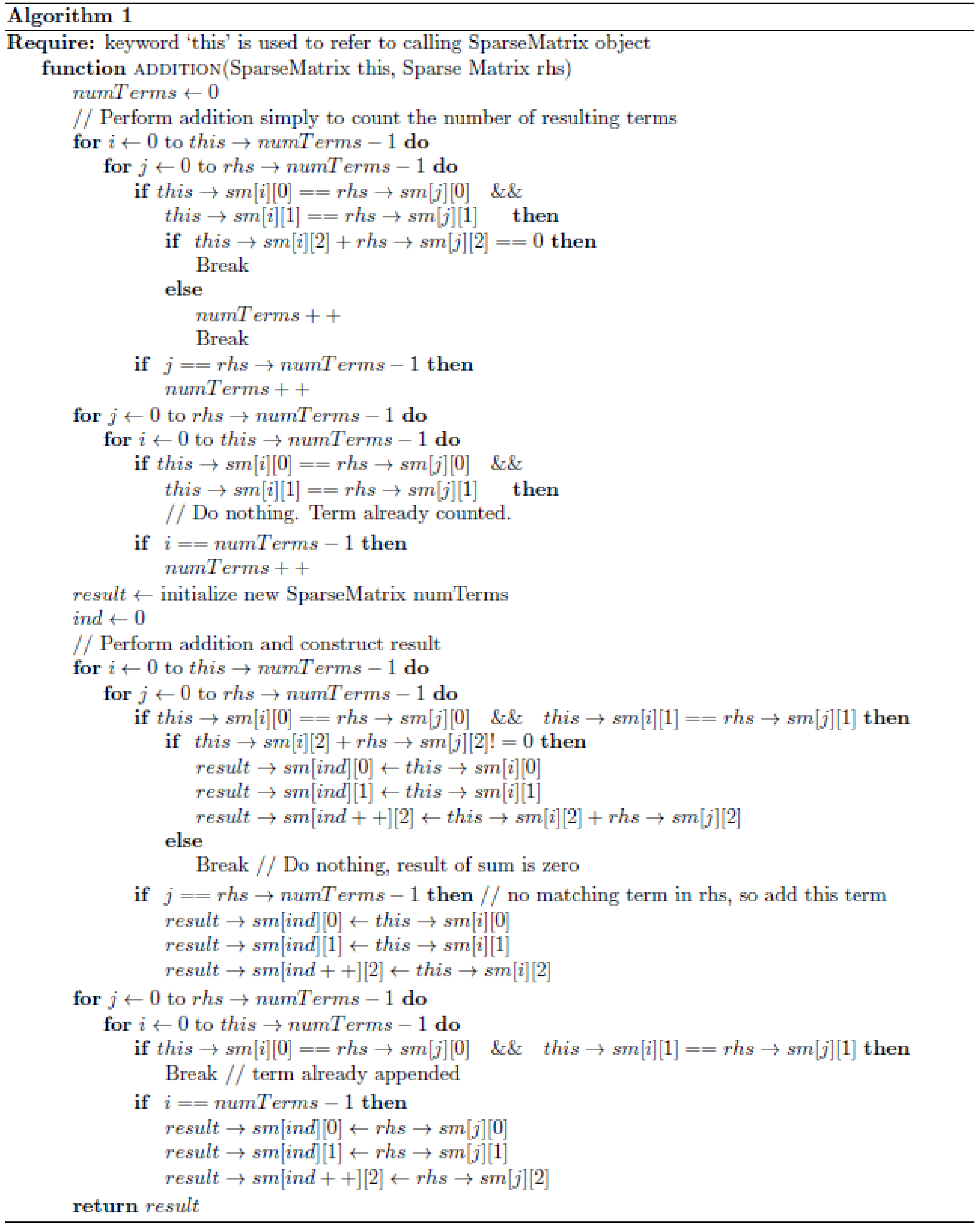
Build (initialize) jump table RowStart, based on counts of elements per column.

Use jump table. Determine each entries index into the transposed matrix using jump table

GEORGETOWN UNIVERSITY

Array Implementation: Addition (version 1)

$$\begin{matrix}(0, & 2, & 9)\\(2, & 3, & 2)\\(2, & 4, & 1)\end{matrix} + \begin{matrix}(0, & 1, & 3)\\(2, & 4, & 2)\end{matrix} = \begin{matrix}(0, & 2, & 9)\\(2, & 3, & 2)\\(2 & 4, & 1)\\(0, & 1, & 3)\end{matrix}$$



Algorithm 1

```
Require: keyword 'this' is used to refer to calling SparseMatrix object
  function ADDITION(SparseMatrix this, Sparse Matrix rhs)
    $numTerms \leftarrow 0$
    // Perform addition simply to count the number of resulting terms
    for $i \leftarrow 0$ to $this \rightarrow numTerms - 1$ do
      for $j \leftarrow 0$ to $rhs \rightarrow numTerms - 1$ do
        if $this \rightarrow sm[i][0] == rhs \rightarrow sm[j][0]$ &&
          $this \rightarrow sm[i][1] == rhs \rightarrow sm[j][1]$ then
          if $this \rightarrow sm[i][2] + rhs \rightarrow sm[j][2] == 0$ then
            Break
          else
            $numTerms++$
            Break
        if $j == rhs \rightarrow numTerms - 1$ then
          $numTerms++$
    for $j \leftarrow 0$ to $rhs \rightarrow numTerms - 1$ do
      for $i \leftarrow 0$ to $this \rightarrow numTerms - 1$ do
        if $this \rightarrow sm[i][0] == rhs \rightarrow sm[j][0]$ &&
          $this \rightarrow sm[i][1] == rhs \rightarrow sm[j][1]$ then
          // Do nothing. Term already counted.
        if $i == numTerms - 1$ then
          $numTerms++$
    $result \leftarrow$ initialize new SparseMatrix numTerms
    $ind \leftarrow 0$
    // Perform addition and construct result
    for $i \leftarrow 0$ to $this \rightarrow numTerms - 1$ do
      for $j \leftarrow 0$ to $rhs \rightarrow numTerms - 1$ do
        if $this \rightarrow sm[i][0] == rhs \rightarrow sm[j][0]$ && $this \rightarrow sm[i][1] == rhs \rightarrow sm[j][1]$ then
          if $this \rightarrow sm[i][2] + rhs \rightarrow sm[j][2]! = 0$ then
            $result \rightarrow sm[ind][0] \leftarrow this \rightarrow sm[i][0]$
            $result \rightarrow sm[ind][1] \leftarrow this \rightarrow sm[i][1]$
            $result \rightarrow sm[ind++][2] \leftarrow this \rightarrow sm[i][2] + rhs \rightarrow sm[j][2]$
          else
            Break // Do nothing, result of sum is zero
        if $j == rhs \rightarrow numTerms - 1$ then // no matching term in rhs, so add this term
          $result \rightarrow sm[ind][0] \leftarrow this \rightarrow sm[i][0]$
          $result \rightarrow sm[ind][1] \leftarrow this \rightarrow sm[i][1]$
          $result \rightarrow sm[ind++][2] \leftarrow this \rightarrow sm[i][2]$
    for $j \leftarrow 0$ to $rhs \rightarrow numTerms - 1$ do
      for $i \leftarrow 0$ to $this \rightarrow numTerms - 1$ do
        if $this \rightarrow sm[i][0] == rhs \rightarrow sm[j][0]$ && $this \rightarrow sm[i][1] == rhs \rightarrow sm[j][1]$ then
          Break // term already appended
        if $i == numTerms - 1$ then
          $result \rightarrow sm[ind][0] \leftarrow rhs \rightarrow sm[j][0]$
          $result \rightarrow sm[ind][1] \leftarrow rhs \rightarrow sm[j][1]$
          $result \rightarrow sm[ind++][2] \leftarrow rhs \rightarrow sm[j][2]$
    return $result$
```

Array Implementation: Addition (version 2)

```
Algorithm 1
Require: keyword 'this' is used to refer to calling SparseMatrix object
  function ADDITION(SparseMatrix this, Sparse Matrix rhs)
    $numTerms \leftarrow 0$
    // Perform addition simply to count the number of resulting terms
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < this \rightarrow numTerms \&\& j < rhs \rightarrow numTerms$ do
      $thisOrder \leftarrow this \rightarrow sm[i][0] * this \rightarrow numCols + this \rightarrow sm[i][1]$
      $rhsOrder \leftarrow rhs \rightarrow sm[i][0] * rhs \rightarrow numCols + rhs \rightarrow sm[i][1]$
      if $thisOrder < rhsOrder$ then // this term is not in rhs
        $numTerms++$
        $i++$
      if $thisOrder > rhsOrder$ then // rhs term is not in this
        $numTerms++$
        $j++$
      if $thisOrder == rhsOrder$ then // matching terms
        if $this \rightarrow sm[i][2] + rhs \rightarrow sm[i][2]! = 0$ then
          $numTerms++$
        $i++$
        $j++$
    while $i < this \rightarrow numTerms$ do
      $numTerms++$
      $i++$
    while $j < rhs \rightarrow numTerms$ do
      $numTerms++$
      $j++$
    $result \leftarrow$ initialize new SparseMatrix numTerms
    $ind \leftarrow 0$
    // Perform addition and construct result
    while $i < this \rightarrow numTerms \&\& j < rhs \rightarrow numTerms$ do
      $thisOrder \leftarrow this \rightarrow sm[i][0] * this \rightarrow numCols + this \rightarrow sm[i][1]$
      $rhsOrder \leftarrow rhs \rightarrow sm[i][0] * rhs \rightarrow numCols + rhs \rightarrow sm[i][1]$
      if $thisOrder < rhsOrder$ then // this term is not in rhs
        $result \rightarrow sm[ind][0] \leftarrow this \rightarrow sm[i][0]$
        $result \rightarrow sm[ind][1] \leftarrow this \rightarrow sm[i][1]$
        $result \rightarrow sm[ind++][2] \leftarrow this \rightarrow sm[i++][2]$
      if $thisOrder > rhsOrder$ then // rhs term is not in this
        $result \rightarrow sm[ind][0] \leftarrow rhs \rightarrow sm[j][0]$
        $result \rightarrow sm[ind][1] \leftarrow rhs \rightarrow sm[j][1]$
        $result \rightarrow sm[ind++][2] \leftarrow rhs \rightarrow sm[j++][2]$
      if $thisOrder == rhsOrder$ then // matching terms
        if $this \rightarrow sm[i][2] + rhs \rightarrow sm[i][2]! = 0$ then
          $result \rightarrow sm[ind][0] \leftarrow this \rightarrow sm[i][0]$
          $result \rightarrow sm[ind][1] \leftarrow this \rightarrow sm[i][1]$
          $result \rightarrow sm[ind++][2] \leftarrow this \rightarrow sm[i++][2] + rhs \rightarrow sm[j++][2]$
    while $i < this \rightarrow numTerms$ do
      $result \rightarrow sm[ind][0] \leftarrow this \rightarrow sm[i][0]$
      $result \rightarrow sm[ind][1] \leftarrow this \rightarrow sm[i][1]$
      $result \rightarrow sm[ind++][2] \leftarrow this \rightarrow sm[i++][2]$
    while $j < rhs \rightarrow numTerms$ do
      $result \rightarrow sm[ind][0] \leftarrow rhs \rightarrow sm[j][0]$
      $result \rightarrow sm[ind][1] \leftarrow rhs \rightarrow sm[j][1]$
      $result \rightarrow sm[ind++][2] \leftarrow rhs \rightarrow sm[j++][2]$
    return $result$
```

Sparse Matrix: Array Implementation

- Try Multiplication as an exercise at home

- Notes:
  - The result of the multiplication of two sparse matrices may not be sparse!
    - Observe sparse matrix representations are not necessarily efficient when matrices are not sparse.
    - The goal of sparse matrix representation is to improve efficiency. When does the sparse matrix representation lose its benefits?

Sparse Matrices: Linked List Implementation

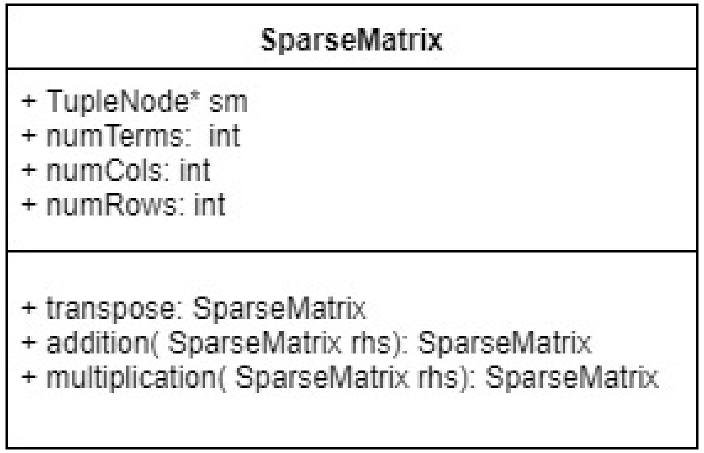
- The use of sparse matrix (array) improved efficiency (as compared to standard matrix) when the number of non-zero terms was small.
  - Transpose: Θ(numTerms + numCols) vs. Θ(numCols * numRows)
  - Addition: Θ(numTerms) vs. Θ(numCols * numRows)

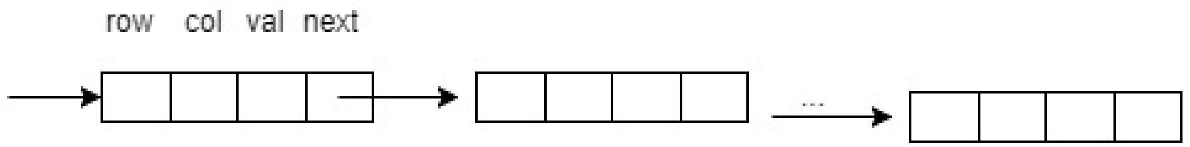
- The array implementation is intuitive, but some memory allocation constraints (knowing the size of a resulting sparse matrix) seemingly required unnecessary computational steps
  - Addition: Size of resulting sum is unknown! Iterate over each sparse matrix simply to determine the size of resulting sum for allocation.
  - Transpose: The size of the resulting transposed matrix is known. However, when copying terms to the transposed sparse matrix, the index where to insert each term is not known without some extra computational overhead. That is, to maintain an ordered structure, extra computation steps were needed to organize the insertion into the new array.

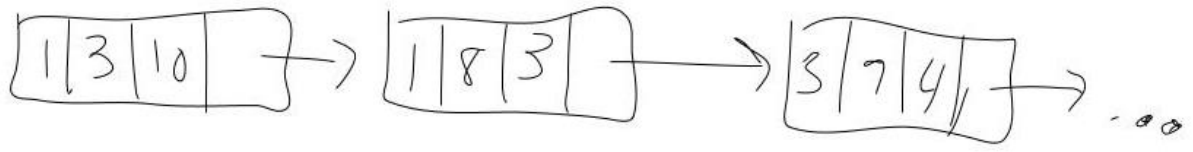
- We have investigated the implementation of a sparse matrix using parallel arrays (or similarly one could use an array of tuples)
  - There are many variations to the use of arrays, parallel arrays, arrays of tuples, etc. And there are many variations on the standard matrix operations (as we have seen thus far).
  - Another design approach: Is there a benefit to using a chaining strategy as opposed to arrays?

Chaining Design for Sparse Matrix

- Option: Design each node to contain 3-tuple and a pointer to the next non-zero term in the matrix (in a row major scan).
  - Efficiency note: We will not need to determine the size of the resulting sum before we can start adding terms. (Linked lists can grow dynamically!)


SparseMatrix
+ TupleNode* sm
+ numTerms: int
+ numCols: int
+ numRows: int
+ transpose: SparseMatrix
+ addition(SparseMatrix rhs): SparseMatrix
+ multiplication(SparseMatrix rhs): SparseMatrix





Sparse Matrix: Chaining

- Example pseudocode for addition
  - Time: Similar to array implementation but with one less pass over each a and b (for allocation size). Thus saving about ½ the computational steps.
  - Memory: one extra pointer must be stored per term

```
Algorithm 1
Require: keyword 'this' is used to refer to calling SparseMatrix object
  function ADDITION(SparseMatrix this, Sparse Matrix rhs)
    $result \leftarrow$ initialize new SparseMatrix
    $thisIter \leftarrow this \rightarrow sm$
    $rhsIter \leftarrow rhs \rightarrow sm$
    while $thisIter \neq NULL \quad \&\& \quad rhsIter \neq NULL$ do
      $thisOrder \leftarrow thisIter \rightarrow row * thisIter \rightarrow numCols + thisIter \rightarrow col$
      $rhsOrder \leftarrow rhsIter \rightarrow row * rhsIter \rightarrow numCols + rhsIter \rightarrow col$
      if $thisOrder < rhsOrder$ then // this term is not in rhs
        $result.addTermToBack(thisIter)$
        $thisIter \leftarrow thisIter \rightarrow next$
      if $thisOrder > rhsOrder$ then // rhs term is not in this
        $result.addTermToBack(rhsIter)$
        $rhsIter \leftarrow rhsIter \rightarrow next$
      if $thisOrder == rhsOrder$ then // matching terms
        $result.addTermToBack(rhsIter)$
        $rhsIter \leftarrow rhsIter \rightarrow next$
        $thisIter \leftarrow thisIter \rightarrow next$
    while $thisIter \neq NULL$ do
      $result.addTermToBack(thisIter)$
      $thisIter \leftarrow thisIter \rightarrow next$
    while $rhsIter \neq NULL$ do
      $result.addTermToBack(rhsIter)$
      $rhsIter \leftarrow rhsIter \rightarrow next$
    return $result$
```

Sparse Matrix: Chaining

- Example pseudocode for a **transpose method**
  - Time and Space analysis?
  - Can we do better?

Algorithm 1

```
Require: keyword ‘this’ is used to refer to calling SparseMatrix object
  function TRANSPOSE(SparseMatrix this)
    $result \leftarrow$ initialize new SparseMatrix
    $thisIter \leftarrow this \rightarrow sm$
    while $thisIter \neq NULL$ do
      $result.insertInOrder(thisIter.swapRowAndCol())$ // Time Complexity?
      $thisIter \leftarrow thisIter \rightarrow next$
    return $result$
```

Sparse Matrix: Chain

- Transpose (version 2)
  - Note:
    - We used a jump table to construct a FAST (linear time) transpose for the array implementation
    - BUT … Linked Lists (Chains) do not permit random (direct) access.
    - One solution for Chain implementation which maintains similar linear time:
      1. Construct Array Representation of Sparse Matrix // O(numTerms)
      2. Perform Matrix version of transpose // O(numTerms + numCols)
      3. Construct Chain Sparse Matrix // O(numTerms)

      TOTAL: O(numTerms + numCols)

      – Side Note: Here we solve a new problem by mapping the problem to a different domain for which a solution already exists. As long as the mapping (and inverse) has a time complexity less than the computed solution, this may be a reasonable approach.

Matrix Representation Comparison

Operation	Matrix	Sparse Matrix (array)	Sparse Matrix (chain)
Addition (time)	$\Theta(numCol * numRow)$	$\Theta(numTermsA + numTermsB)$	$\Theta(numTermsA + numTermsB)$
Transpose (time)	$\Theta(numCol * numRow)$	$\Theta(numTerms + numCols)$	$\Theta(numTerms + numCols)$ // assuming copy to array

- Other:
  - Addition: array implementation required twice as many computational steps to allocate memory efficiently
  - Transpose: Sparse Matrix may also alleviate some cache delays as compared to Matrix

GEORGETOWN UNIVERSITY

Project: Polynomials

- Reminder: With the sparse matrix structure*, we faced many structural design questions and subsequent algorithmic design questions, both of which affected efficiency.* You will face similar design questions in your polynomial project.

- Design a Representation (Data Structure) for Polynomials
  - Goals:
    - Polynomial evaluation
    - Polynomial arithmetic

- Class Project: Design Questions and Goals.
  - Linked Chain vs Array Implementation?
  - Goal: An efficient Solution (time and space)
    - Algorithmic improvements for basic operations
    - How can we increase efficiency: reduce computational complexity?
    - When you make a design decision, document the reason why and justify in the cover letter.
    - Use average and worst cases to make design decisions (not best case)