```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% File:   spmatS2.tu
% Author: Gunnar Gotshalks
% Date:   1997 January 21
%
% This file contains Part 2/2 of the sparse matrix class solution.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constructor and destructor operations.
%
% Initialize  must be the first procedure called when using a sparse matrix.
% Initial conditions for an empty matrix are established.
%
% Precondition:  True
% Postcondition: count = maxRow = maxCol = 0 and
%     rowList(1..MaxRowsCols) = nil = colList(1..MaxRowsCols)

procedure Initialize()
    for i : 1 .. MaxRowsCols
        rowList (i) := nil   % All rows are empty
        colList (i) := nil   % All columns are empty
    end for
    count := 0               % No elements in the matrix
    maxRow := 0              % Set to no rows
    maxCol := 0              % Set to no columns
end Initialize

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finalize must be the last procedure called when using a sparse matrix.
% Its purpose is to return all matrix elements to the heap.  It uses direct
% knowledge of the implementation.
%
% Precondition:  The matrix has been initialized.
% Postcondition: All matrix elements are finalized and freed.  The matrix is
%     initialized.
%
% Loop invariant: for i:1..count+1 :: freedElement(1..i-1)

procedure Finalize()
    var prevElem : ^MatrixElement  % Element to finalize and free
    var nextElem : ^MatrixElement  % Element following the one to free

  if Debug.flag2 then put "Count before finalize is ", count end if

    for row : 1 .. maxRow
        nextElem := rowList (row)
        loop
            exit when nextElem = nil
            prevElem := nextElem
            nextElem := nextElem->GetNextInRow()
            prevElem->Finalize()
            free prevElem
            count -= 1
        end loop
    end for

  if Debug.flag2 then put "Count after Finalize is ", count end if

    Initialize()
end Finalize

%%%%%%%%%%%%%%%% Using iterator version of Finalize.
% It works because the variable nextElem (in the class) keeps track of the
% element following selfElem.

procedure Finalize2()
    var selfElem : ^MatrixElement  % Element to finalize and free

    self->InitIterationByRow()
    selfElem := self->NextIterRowElem()
    loop
        exit when selfElem ~= nil
        selfElem->Finalize()
        free selfElem
        selfElem := self->NextIterRowElem()
    end loop

  if Debug.flag2 then put "Count after Finalize is ", count end if

    Initialize()
end Finalize2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Debug put procedures to minimize the distraction in the operations.  Also to
% standardize the output format for debug statments.

procedure Debug1(opName, oper1 : string, operVal1 : int,
                 prevElem : ^MatrixElement)
    if Debug.flag0 then
        put "  ", opName, " ", oper1, " = ", operVal1 ..
        if prevElem = nil then put "; prevElem = nil"
                          else put "; prevElem not nil"
        end if
    end if
end Debug1

procedure Debug2(opName, oper1 : string, operVal1 : int,
                 oper2 : string, operVal2 : int)
    if Debug.flag0 then
          put "  ", opName, " ", oper1, " = ", operVal1,
              "; ", oper2, " = ", operVal2
    end if
end Debug2

procedure Debug3(opName : string)
   if Debug.flag0 then
        put "  Exit ", opName, " count = ",count,"; maxRow = ",maxRow,
            "; maxCol = ",maxCol
   end if
end Debug3

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Utility operations
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The following operations support inserting an element into a matrix.  Only
% Insert is exported.
%
% FindPrevRowElem searches the row, keyRow, for the matrix element with the
% highest column number that is less than valueCol.  Result is prevElem, a
% pointer to that element.  prevElem = nil means there is no such element
% in the row.
%
% Precondition:  Matrix has been initialized.
% Postcondition:  prevElem = nil
```

```
%        or (     prevElem->nextInRow->col < valueCol )
%            and  prevElem is an element in row keyRow)
%
% Loop invariant: for existing row elements :: theElem(1..n-1)->col < valueCol

function FindPrevRowElem (keyRow, valueCol : int,
                          var theElem : ^MatrixElement) : ^MatrixElement
  if Debug.flag0 then
      put "  FindPrevRowElem keyRow = ", keyRow,
          "; valueCol = ", valueCol
  end if

    var prevElem : ^MatrixElement := nil
    theElem := rowList(keyRow)

    loop
        exit when theElem = nil or theElem->GetCol() >= valueCol
        prevElem := theElem
        theElem := theElem->GetNextInRow()
    end loop

% Set theElem to nil if it is not pointing to the element being searched for.

    if theElem ~= nil and theElem->GetCol() ~= valueCol then
        theElem := nil
    end if

  if Debug.flag0 then
      if prevElem = nil then
          put "  Exit FindPrevRowElem prevElem = nil"
      else
          put "  Exit FindPrevRowElem prevElem not nil"
      end if
  end if

    result prevElem
end FindPrevRowElem

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% FindPrevColElem searches the column, keyCol, for the matrix element with the
% highest row number that is less than valueRow.  Result is prevElem, a
% pointer to that element.  prevElem = nil means there is no such element
% in the column.
%
% Precondition:  Matrix has been initialized.
% Postcondition:  prevElem = nil
%        or (     prevElem->nextInCol->col < valueCol )
%            and  prevElem is an element in column keyCol)
%
% Loop invariant:
%       for existing column elements :: theElem(1..n-1)->row < valueRow

function FindPrevColElem(keyCol, valueRow : int,
                     var theElem : ^MatrixElement) : ^MatrixElement
  if Debug.flag0 then
      put "  FindPrevColElem keyCol = ", keyCol,
          "; valueRow = ", valueRow
  end if

    var prevElem : ^MatrixElement := nil
    theElem := colList(keyCol)

    loop
        exit when theElem = nil or theElem->GetRow() >= valueRow
        prevElem := theElem
        theElem := theElem->GetNextInCol()
    end loop

% Set theElem to nil if it is not pointing to the element being searched for.

    if theElem ~= nil and theElem->GetRow() ~= valueRow then
        theElem := nil
    end if

  if Debug.flag0 then
      if prevElem = nil then
          put "  Exit FindPrevRowElem prevElem = nil"
      else
          put "  Exit FindPrevRowElem prevElem not nil"
      end if
  end if

    result prevElem
end FindPrevColElem

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Insert into the row, row, a matrix element, newElem, given a pointer to
% its predecessor element, prevElem.  prevElem = nil means put at the
% front of the list as there is no predecessor element.
%
% Precondition:  (prevElem is in the row, row, or prevElem = nil)
%       and  1 <= newElem->row <= MaxRowsCols
%       and  1 <= newElem->col <= MaxRowsCols
%
% Postcondition: newElem is in rowList(row)
%       prevElem = nil  => rowList'(row) = newElem
%                          and newElem->nextColElem = rowList(row)
%   and prevElem ~= nil => newElem->nextColElem = prevElem->nextColElem
%                          and prevElem->nextColElem' = newELem

procedure InsertInRow(row : int, prevElem, newElem : ^MatrixElement)
  Debug1("InsertInRow","row",row, prevElem)
    if prevElem = nil then   % Insert at head of list
        newElem->SetNextInRow(rowList(row))
        rowList(row) := newElem
    else                     % Insert somewhere after head of list
        newElem->SetNextInRow(prevElem->GetNextInRow())
        prevElem->SetNextInRow(newElem)
    end if
end InsertInRow

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Insert into column, col, a matrix element, newElem, given a pointer to
% its predecessor element, prevElem.  prevElem = nil means put at the
% front of the list as there is no predecessor element.
%
% Precondition:  (prevElem in column, col or prevElem = nil)
%       and  1 <= newElem->row <= MaxRowsCols
%       and  1 <= newElem->col <= MaxRowsCols
%
% Postcondition: newElem is in colList(col)
%       prevElem = nil  => colList(col)' = newElem
%                          and newElem->nextRowElem = colList(col)
%   and prevElem ~= nil => newElem->nextRowElem = prevElem->nextRowElem
%                          and prevElem->nextRowElem' = newELem

procedure InsertInCol(col : int, prevElem, newElem : ^MatrixElement)
  Debug1("InsertInCol","col",col, prevElem)
    if prevElem = nil then   % Insert at head of list
        newElem->SetNextInCol(colList(col))
```

```
        colList(col) := newElem
    else                    % Insert somewhere after head of list
        newElem->SetNextInCol(prevElem->GetNextInCol())
        prevElem->SetNextInCol(newElem)
    end if
end InsertInCol

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Delete in row, row, the matrix element pointed to by prevElem.
% prevElem = nil means delete the first element in the column as there
%is no predecessor element.
%
% Precondition:  (prevElem in row, row, and prevELem->nextInRow ~= nil)
%            or  (prevElem = nil and rowList(row) ~= nil)
%
% Postcondition: prevElem = nil  => rowList(row)' = rowList(row)->nextInRow
%          and   prevElem ~= nil =>
%                   prevElem->nextInCol' = prevElem->nextInRow->nextInRow

procedure DeleteInRow(row : int, prevElem : ^MatrixElement)
  Debug1("DeleteInRow","row",row, prevElem)
    if prevElem = nil then   % Delete at head of list
        rowList(row) := rowList(row)->GetNextInRow()
    else                    % Delete somewhere after head of list
        prevElem->SetNextInRow(prevElem->GetNextInRow()->GetNextInRow())
    end if
end DeleteInRow

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Delete in column, col, the matrix element pointed to by prevElem.
% prevElem = nil means delete the first element in the column as there
%is no predecessor element.
%
% Precondition:  (prevElem in column, col, and prevELem->nextInCol ~= nil)
%            or  (prevElem = nil and colList(col) ~= nil)
%
% Postcondition: prevElem = nil  => colList(col)' = colList(col)->nextInCol
%          and   prevElem ~= nil =>
%                   prevElem->nextinCol' = prevElem->nextInCol->nextInCol

procedure DeleteInCol(col : int, prevElem : ^MatrixElement)
  Debug1("DeleteInCol","col",col, prevElem)
    if prevElem = nil then   % Delete at head of list
        colList(col) := colList(col)->GetNextInCol()
    else                    % Delete somewhere after head of list
        prevElem->SetNextInCol(prevElem->GetNextInCol()->GetNextInCol())
    end if
end DeleteInCol

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Update metadata count, maxRow and maxCol for Insert
%
% Precondition:  An element has just been inserted in both row and
%       column lists.
% Postcondition: count' = count+1
%           and  maxRow' = max(maxRow,row)
%           and  maxCol' = max(maxCol,col)

procedure UpdateMetaForInsert(row, col : int)
  Debug2("UpdateMetaForInsert","row",row,"col",col)
    count += 1
    if maxRow < row then maxRow := row end if
    if maxCol < col then maxCol := col end if
  Debug3("UpdateMetaForInsert")
end UpdateMetaForInsert

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Update metadata count, maxRow and maxCol for Delete
%
% Precondition:  An element has just been deleted from both row and
%       column lists.
% Postcondition: count' = count - 1
%           and  for i:maxRow'+1..MaxRowsCols :: rowList(i) = nil
%           and  (maxRow = 0 or rowList(maxRow) ~= nil)
%           and  for i:maxCol'+1..MaxRowsCols :: colList(i) = nil
%           and  (maxCol = 0 or colList(maxCol) ~= nil)

procedure UpdateMetaForDelete(row, col : int)
  Debug2("UpdateMetaForInsert","row",row,"col",col)
    count -= 1

  % Loop invariants:
  %     for i : maxRow+1..MaxRowsCols :: rowList(i) = nil
  %     for i : maxCol+1..MaxRowsCols :: colList(i) = nil

    loop exit when maxRow = 0 or rowList(maxRow) ~= nil
        maxRow -= 1
    end loop
    loop exit when maxCol = 0 or colList(maxCol) ~= nil
        maxCol -=1
    end loop
  Debug3("UpdateMetaForInsert")
end UpdateMetaForDelete

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Depending upon its input parameters, Insert either inserts a new element,
% replaces the data value for an existing element, or deletes an existing
% element.
%
% Precondition:  Matrix has been initialized.
%       and  1 <= row <= MaxRowsCols
%       and  1 <= col <= MaxRowsCols
%
% Postcondition:
%       if data ~= 0 and  existsElement(row,col) then data replaces the value
%       if data ~= 0 and ~existsElement(row,col) then new element is inserted
%       if data  = 0 and  existsElement(row,col) then the element is deleted
%       if data  = 0 and ~existsElement(row,col) then skip

procedure Insert(row, col, data : int)
    var prevRowElem : ^MatrixElement  % Pointer to element to insert after
                                      % in a row list.
    var prevColElem : ^MatrixElement  % Pointer to element to insert after
                                      % in a column list.
    var theElem     : ^MatrixElement  % Pointer to <row,col> element, if
                                      % it exists, nil otherwise.

  if Debug.flag0 then
       put "  Insert row = ", row, "; column = ",col, "; data = ",data
  end if

    var newElem  : ^MatrixElement     % Element to insert into the matrix.

% Find the pointer to previous element in the row list and in the column list.
% This is a pointer to element after which the potentially new element will
% will be inserted.  Also, if an element already exits at the input row and
% column, get a pointer to that element.

    prevRowElem := FindPrevRowElem(row, col, theElem)
    prevColElem := FindPrevColElem(col, row, theElem)
```

```
% Check for replacing an existing data value with a new nonzero value.
% Metadata does not change.

    if    data ~= 0 and theElem ~= nil then
        theElem->SetData(data)

% Check for inserting a new element into the matrix.  Metadata cannot be
% updated in either InsertInRow or InsertInCol as they would then not be
% symetric operations.  It cannot be updated in both as that would double
% count the number of inserted elements.

    elsif data ~= 0 and theElem  = nil then
        new newElem
        newElem->Initialize(row, col, data)

        InsertInRow(row, prevRowElem, newElem)
        InsertInCol(col, prevColElem, newElem)
        UpdateMetaForInsert(row,col)

% Check for deleting an existing element, since it is being replaced with
% a zero value.

    elsif data  = 0 and theElem ~= nil then
        DeleteInRow(row, prevRowElem)
        DeleteInCol(col, prevColElem)
        theElem->Finalize()
        free theElem
        UpdateMetaForDelete(row,col)

% Nothing else to do since for the remaining case a zero value is not
% inserted into the matrix.

    end if

  if Debug.flag0 then
      if Debug.flag1 then
          put "Count is ", count, "; maxRow is ", maxRow,
              "; maxCol is ", maxCol
       end if
       put "  Exit Insert"
  end if

end Insert

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GetData returns the value of a data item represented in the array.
% If rowCol = 'r', the search is along a row list.  If rowCol = 'c', the
% search is along a column list.
% Precondition:  1 <= row <= maxRow  and  1 <= col <= maxCol
%            and  rowCol in {'r','c'}
% Postcondition:  data at element <row,col>

function GetData(row, col : int, rowCol : char) : int
    if rowCol = 'r' then
        var currElem : ^MatrixElement := rowList(row)

% Loop invariant:
%   for existing row elements :: currElem(1..n-1)->col < valueCol

        loop
            exit when currElem = nil or currElem->GetCol() >= col
            currElem := currElem->GetNextInRow()
        end loop
        if currElem = nil then result 0
                          else result currElem->GetData()
        end if
    else
        var currElem : ^MatrixElement := colList(col)

% Loop invariant:
%   for existing column elements :: currElem(1..n-1)->row < valueRow

        loop
            exit when currElem = nil or currElem->GetRow() >= row
            currElem := currElem->GetNextInCol()
        end loop

        if currElem = nil then result 0
                          else result currElem->GetData()
        end if
    end if
end GetData

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ExistsElement checks if an element is physically present.
% If rowCol = 'r', the search is along a row list.  If rowCol = 'c', the
% search is along a column list.
%
% Precondition:  1 <= row <= maxRow  and  1 <= col <= maxCol
%            and  rowCol in {'r','c'}
% Postcondition: result = true  => element physically exists
%            and  result = false => element does not physically exist
%

function ExistsElement(row, col : int, rowCol : char) : boolean
    if rowCol = 'r' then
        var currElem : ^MatrixElement := rowList(row)

% Loop invariant:
%   for existing row elements :: currElem(1..n-1)->col < valueCol

        loop
            exit when currElem = nil or currElem->GetCol() >= col
            currElem := currElem->GetNextInRow()
        end loop

        if currElem = nil then result false
                          else result currElem->GetCol() = col
        end if
    else
        var currElem : ^MatrixElement := colList(col)

% Loop invariant:
%   for existing column elements :: currElem(1..n-1)->row < valueRow

        loop
            exit when currElem = nil or currElem->GetRow() >= row
            currElem := currElem->GetNextInCol()
        end loop

        if currElem = nil then result false
                          else result currElem->GetRow() = row
        end if
    end if
end ExistsElement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Put prints the sparse matrix in a rectangular format with right justified
```

```
%"**" for the nonexistent elements.
%
% Precondition:  Matrix has been initialized.
% Postcondition: Matrix is displayed in rectangular format, maxRow x maxCol,
% with. colHeadings(1..maxCol) and rowHeadings(1..maxRow).

procedure Put()
    var rowIndex, colIndex : int % Loop counters for row and column
    var elem : ^MatrixElement % Element to show

    % Check for empty matrix

    if maxRow < 1 then
        put "Empty Matrix"
    end if

    % Output the column headings.

    put "     " ..
    for i : 1 .. maxCol ; put i : 5 .. ; end for

    put "\n     " ..
    for i : 1 .. maxCol ; put "  ___" .. ; end for
    put ""

% Outer loop displays a row at a time.
% Loop Invariant:  display(rows(1..rowIndex-1))

    rowIndex := 1
    loop
        exit when rowIndex > maxRow
        put rowIndex : 3, "| " ..
        colIndex := 1
        elem := rowList(rowIndex)

    % Inner loop displays all the elements in a row.
    % Loop Invariant:  display(columnElements(1..colIndex-1)

        loop
            exit when colIndex > maxCol
            if elem = nil then
                put "   **" ..
            elsif elem->GetCol() = colIndex then
                put elem->GetData() : 5 ..
                elem := elem->GetNextInRow()
            else
                put "   **" ..
            end if
            colIndex += 1
        end loop
        put ""
        rowIndex += 1
    end loop

end Put

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Support functions for appending a new element to the "end" of the row
% order of a matrix.

% InitAppend creates a temporary record containing metadata pointing to the
% ends of the each column list, the end of the current row and what row it is.
%
% Precondition:  True
% Postcondition: A new initialized record of type ListEndPtrs is created

type ListEndPtrs : record
        lastInRow : ^MatrixElement
        lastInCol : array 1..MaxRowsCols of ^MatrixElement
        lastRow   : int
    end record

function InitAppend() : ^ListEndPtrs
    var endPtrs : ^ListEndPtrs
    new endPtrs
    endPtrs->lastInRow := nil
    for i : 1..MaxRowsCols
        endPtrs->lastInCol(i) := nil
    end for
    result endPtrs
end InitAppend

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% When appending is finished then the record created by InitAppend needs to be
% freed.  Var is required because nil is stored in a pointer when an object
% is freed.
%
% Precondition:  endPtrs has been created.
% Postcondition: The object endPtrs points to has been freed.

procedure FinalizeAppend(var endPtrs : ^ListEndPtrs)
   free endPtrs
end FinalizeAppend

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% AppendNewElement uses self metadata to add the new matrix element to the
% end of the appropriate row and column lists.
%
% Precondition:  The element is to be added to end of its row and column lists.
%           Elements to be appended arrive in row order.
%       1 <= theRow <= MaxRowsCols  and  1 <= theCol <= MaxRowsCols
% Postcondition: The element is added to the end of its row and column lists.
%       maxRow' = max(maxRow,theRow)  and  maxCol' = max(maxCol,theCol)
%   and count' = count + 1

procedure AppendNewElement(theRow, theCol, theData : int,
                           lastPtrs : ^ListEndPtrs)
    var newElem : ^MatrixElement
    new newElem
    newElem->Initialize(theRow, theCol, theData)

    if maxRow = theRow then  % Append to an existing row element.
        lastPtrs->lastInRow->SetNextInRow(newElem)
    else                     % Start a new row.
        rowList(theRow) := newElem
        maxRow := theRow
    end if
    lastPtrs->lastInRow := newElem

% Always add to the end of a different column.

    if lastPtrs->lastInCol(theCol) ~= nil then
        lastPtrs->lastInCol(theCol)->SetNextInRow(newElem)
    else colList(theCol) := newElem
    end if
    lastPtrs->lastInCol(theCol) := newElem

% Update the remaining metadata variables.
```

```
    count += 1
    if maxCol < theCol then maxCol := theCol end if

end AppendNewElement

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The following are support procedures for Sum. Each procedure handles a
% special case of operands being the same matrix.
%
% The matrix is to be summed with itself.
% Postcondition:  self' = self + self
%
% This algorithm requires no insertions into the result.  The entries in
% self are all doubled in value.
%
% Loop invariant:  RowOrderElements(1..n-1) have data' = 2*data

procedure AddSelfToSelf()
    var matElem : ^MatrixElement

    InitIterationByRow()
    matElem := NextIterRowElem()
    loop exit when matElem = nil
        matElem->SetData(2 * matElem->GetData())
        matElem := NextIterRowElem()
    end loop
end AddSelfToSelf

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For adding the same matrix to itself and the sum is not that matrix.
% Postcondition: self' = theMatrix + theMatrix
%
% The sum can be finalized.  We use only one iterator to go through the
% matrix, dobule each data value and insert it into the sum.

procedure AddMatrixTwice(theMatrix : ^SparseMatrix)
    var dataSum  : int            % The sum of data values in self and matrix2
    var matElem  : ^MatrixElement % An element in the matrix1
    var lastPtrs : ^ListEndPtrs   % Temporary record of the ends of row
                                  % and column lists for appending new
                                  % elements.
    Finalize()
    lastPtrs := InitAppend()
    theMatrix->InitIterationByRow()
    matElem  := theMatrix->NextIterRowElem()

% Loop Invariant: RowOrderElementsSumMatrix(1..n-1)->data =
%                               2*RowOrderElementsTheMatrix(1..n-1)

    loop exit when matElem = nil
        AppendNewElement(matElem->GetRow(), matElem->GetCol(),
            2 * matElem->GetData(), lastPtrs)
        matElem := theMatrix->NextIterRowElem()
    end loop
    FinalizeAppend(lastPtrs)

end AddMatrixTwice

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The final case is when all the matrices are different.  Sum is finalized
% before the two matrices are added.
% Postcondition: sum' = matrix1 + matrix2

procedure AddDiffMatrices(matrix1,matrix2 : ^SparseMatrix)
    var dataSum  : int            % The sum of data values in self and matrix2
    var matElem1 : ^MatrixElement % An element in the matrix1
    var matElem2 : ^MatrixElement % An element in matrix2
    var lastPtrs : ^ListEndPtrs   % Temporary record of the ends of row
                                  % and column lists for appending new
                                  % elements.
    Finalize()
    lastPtrs := InitAppend()
    matrix1->InitIterationByRow()
    matElem1  := matrix1->NextIterRowElem()
    matrix2->InitIterationByRow()
    matElem2 := matrix2->NextIterRowElem()

% First loop merges the matrix elements from matrix1 and matrix2 as long
% as both contain more elements to merge.  The body of the loop examines
% each of the possible cases of <,=,> for row and column indices for
% matrix1 and matrix2.  First by checking on row index then if both elements
% are in the same row, checking the column index.
%
% Loop invariant: RowOrderElementsMatrix1(1..p-1) and
%       RowOrderElementsMatrix2(1..q-1) have been summed creating sum
%       matrix RowOrderElements(1..n-1).  Each time around the loop at
%       least one element is read from one of matrix1 or matrix2.

    loop exit when matElem1 = nil or matElem2 = nil
        if matElem1->GetRow() < matElem2->GetRow() then
            AppendNewElement(matElem1->GetRow(), matElem1->GetCol(),
                             matElem1->GetData(), lastPtrs)
            matElem1 := matrix1->NextIterRowElem()
        elsif matElem1->GetRow() > matElem2->GetRow() then
            AppendNewElement(matElem2->GetRow(), matElem2->GetCol(),
                             matElem2->GetData(), lastPtrs)
            matElem2 := matrix2->NextIterRowElem()
        elsif matElem1->GetCol() < matElem2->GetCol() then
            AppendNewElement(matElem1->GetRow(), matElem1->GetCol(),
                             matElem1->GetData(), lastPtrs)
            matElem1 := matrix1->NextIterRowElem()
        elsif matElem1->GetCol() > matElem2->GetCol() then
            AppendNewElement(matElem2->GetRow(), matElem2->GetCol(),
                             matElem2->GetData(), lastPtrs)
            matElem2 := matrix2->NextIterRowElem()
        else % Same row and column here, only place addition is done.
            dataSum := matElem1->GetData() + matElem2->GetData()
            if dataSum ~= 0 then
                AppendNewElement(matElem1->GetRow(), matElem1->GetCol(),
                                 dataSum, lastPtrs)
            end if
            matElem1 := matrix1->NextIterRowElem()
            matElem2 := matrix2->NextIterRowElem()
        end if
    end loop

% Add in the left over elements from matrix1.
% Precondition:  sum(1..m) = matrix1(1..q) + matrix2(1..matrix2->count)
%           and  q < matrix1->count
% Postcondition: sum(1..sum->count) = matrix1(1..matrix1->count)
%                                    + matrix2(1..matrix2->count)
% Loop invariant: sum(1..p) = matrix1(1..q) + matrix2(1..matrix2->count)

    loop exit when matElem1 = nil
        AppendNewElement(matElem1->GetRow(), matElem1->GetCol(),
                         matElem1->GetData(), lastPtrs)
        matElem1 := matrix1->NextIterRowElem()
    end loop
```

```
% Add in the left over elements from matrix2.
% Precondition:  sum(1..m) = matrix1(1..matrix1-c>count) + matrix2(1..r)
%           and  r < matrix2->count
% Postcondition: sum(1..sum->count) = matrix1(1..matrix1->count)
%                                   + matrix2(1..matrix2->count)
% Loop invariant: sum(1..p) = matrix1(1..matrix1->count) + matrix2(1..r)

    loop exit when matElem2 = nil
        AppendNewElement(matElem2->GetRow(), matElem2->GetCol(),
                         matElem2->GetData(), lastPtrs)
        matElem2 := matrix2->NextIterRowElem()
    end loop
    FinalizeAppend(lastPtrs)
end AddDiffMatrices

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copy a matrix.  By doing it in row order we can use AppendNewElement instead
% of Insert to achieve and O(n) algorithm.
%
% Precondition:  theMatrix is initialized
% Postcondition: result = copy of theMatrix
%
% Loop invariant: self(1..n-1) = theMatrix(1..n-1)

procedure CopyFrom(theMatrix : ^SparseMatrix)
    var matElem  : ^MatrixElement % An element in theMatrix
    var lastPtrs : ^ListEndPtrs   % Temporary record of the ends of row
                                  % and column lists for appending new
                                  % elements.
    lastPtrs := InitAppend()
    theMatrix->InitIterationByRow()
    matElem := theMatrix->NextIterRowElem()

    loop exit when matElem = nil
        AppendNewElement(matElem->GetRow(), matElem->GetCol(),
                         matElem->GetData(), lastPtrs)
        matElem := theMatrix->NextIterRowElem()
    end loop
end CopyFrom

function Copy() : ^SparseMatrix
    var newMatrix : ^SparseMatrix
    new newMatrix
    newMatrix->Initialize()
    newMatrix->CopyFrom(self)
    result newMatrix
end Copy

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This case handles adding a matrix to the sum.  The sum cannot be finalized.
% Postcondition:  self' = self + otherMatrix
%
% Sum cannot be finalized as we need its data.  To get an O(n) algorithm first
% copy the self matrix into a temporary matrix (this is O(n)), then add using
% AddDiffMatrices (this is O(n)), finally destroy the copy (this is O(n)).

procedure AddMatrixToSelf(otherMatrix : ^SparseMatrix)
    var copyOfSelf : ^SparseMatrix := Copy()
    AddDiffMatrices(copyOfSelf,otherMatrix)
    copyOfSelf->Finalize()
    free copyOfSelf          % Complements new in Copy.
end AddMatrixToSelf

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sum adds two matrices -- matrix1 and matrix2.  The algorithm uses a row
% order iterator to scan through each matrix to merge.  Insertion of the
% appropriate sum elements is done directly (without using Insert) using our
% ability to access all the metadata of the sum matrix.  The caller, self,
% does not have to be empty at the time of the call as this operation
% finalizes self.
%
% Precondition: self, matrix1 and matrix2 have been initialized.
% Postcondition: self = matrix1 + matrix2

procedure Sum(matrix1, matrix2 : ^SparseMatrix)

% The following are the cases that require different algorithms depending
% upon which of the three matrices are same as other matrices.  If self is
% one of the input operands then self should not be finalized.

    if    self = matrix1 and self = matrix2 then AddSelfToSelf()
    elsif self = matrix1                    then AddMatrixToSelf(matrix2)
    elsif self = matrix2                    then AddMatrixToSelf(matrix1)
    elsif matrix1 = matrix2                 then AddMatrixTwice(matrix1)
    else  AddDiffMatrices(matrix1,matrix2)
    end if
end Sum


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Support functions to insert of a copy a matrix element for Sum2
%
% Precondition:  True
% Postcondition: The appropriate element is inserted into theMatrix.

procedure InsertCopy(theElem : ^MatrixElement, theMatrix : ^SparseMatrix)
    theMatrix->Insert(theElem->GetRow(), theElem->GetCol(),
                      theElem->GetData())
end InsertCopy

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sum2 adds two matrices -- matrix1 and matrix2.  The algorithm uses a row
% order iterator to scan through each matrix to merge, and uses Insert the
% appropriate sum elements.  The caller, self, does not have to be empty at
% the time of the call as this operation finalizes self.
%
% Precondition: self, matrix1 and matrix2 have been initialized.
% Postcondition: self = matrix1 + matrix2

procedure Sum2(matrix1, matrix2 : ^SparseMatrix)
    var dataSum  : int            % The sum of data values in self and matrix2
    var matElem1 : ^MatrixElement % An element in the matrix1
    var matElem2 : ^MatrixElement % An element in matrix2

    Finalize()
    matrix1->InitIterationByRow()  % Need for both matrix1 = and ~= matrix2.
    matElem1 := matrix1->NextIterRowElem()

% If matrix1 and matrix2 are the same, then we need a special case because
% we can only use the iterator for one of them.  Otherwise the iterator is
% used double the number of times it should be.
%
% Loop Invariant: RowOrderElementsSumMatrix(1..n-1)->data =
%                              2*RowOrderElementsMatrix1(1..n-1)

    if matrix1 = matrix2 then
        loop
            exit when matElem1 = nil
            Insert(matElem1->GetRow(), matElem1->GetCol(),
```

```
                2 * matElem1->GetData())
            matElem1 := matrix1->NextIterRowElem()
        end loop
    else
% First loop merges the matrix elements from matrix1 and matrix2 as long
% as both contain more elements to merge.  The body of the loop examines
% each of the possible cases of <,=,> for row and column indices for
% matrix1 and matrix2.  First by checking on row index then if both elements
% are in the same row, checking the column index.
%
% Loop invariant: RowOrderElementsMatrix1(1..p-1) and
%       RowOrderElementsMatrix2(1..q-1) have been summed creating sum
%       matrix RowOrderElements(1..n-1).  Each time around the loop at
%       least one element is read from one of matrix1 or matrix2.

        matrix2->InitIterationByRow()
        matElem2 := matrix2->NextIterRowElem()

        loop exit when matElem1 = nil or matElem2 = nil
            if matElem1->GetRow() < matElem2->GetRow() then
                InsertCopy(matElem1, self)
                matElem1 := matrix1->NextIterRowElem()
            elsif matElem1->GetRow() > matElem2->GetRow() then
                InsertCopy(matElem2, self)
                matElem2 := matrix2->NextIterRowElem()
            elsif matElem1->GetCol() < matElem2->GetCol() then
                InsertCopy(matElem1, self)
                matElem1 := matrix1->NextIterRowElem()
            elsif matElem1->GetCol() > matElem2->GetCol() then
                InsertCopy(matElem2, self)
                matElem2 := matrix2->NextIterRowElem()
            else % Same row and column here, only place addition is done.
                dataSum := matElem1->GetData() + matElem2->GetData()
                if dataSum ~= 0 then
                    Insert(matElem1->GetRow(), matElem1->GetCol(), dataSum)
                end if
                matElem1 := matrix1->NextIterRowElem()
                matElem2 := matrix2->NextIterRowElem()
            end if
        end loop

% Add in the left over elements from matrix1.
% Precondition:  sum(1..m) = matrix1(1..q) + matrix2(1..matrix2->count)
%           and  q < matrix1->count
% Postcondition: sum(1..sum->count) = matrix1(1..matrix1->count)
%                                   + matrix2(1..matrix2->count)
% Loop invariant: sum(1..p) = matrix1(1..q) + matrix2(1..matrix2->count)

        loop exit when matElem1 = nil
            InsertCopy(matElem1, self)
            matElem1 := matrix1->NextIterRowElem()
        end loop

% Add in the left over elements from matrix2.
% Precondition:  sum(1..m) = matrix1(1..matrix1-c>count) + matrix2(1..r)
%           and  r < matrix2->count
% Postcondition: sum(1..sum->count) = matrix1(1..matrix1->count)
%                                   + matrix2(1..matrix2->count)
% Loop invariant: sum(1..p) = matrix1(1..matrix1->count) + matrix2(1..r)

        loop exit when matElem2 = nil
            InsertCopy(matElem2, self)
            matElem2 := matrix2->NextIterRowElem()
        end loop
    end if

end Sum2
```