Sparse Matrices: An Application of Linked Lists

June 8, 2011

In a sparse matrix, most entries are zeroes:

$$\begin{bmatrix} -1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 4 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 & 6 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Sparse Matrices Are Large

Many problems require *large* (perhaps 100000×100000) sparse matrices.
Store as an ordinary array?

$$100000^2 = 10000000000 = 10 \text{ billion entries}$$

Impractical (usually).

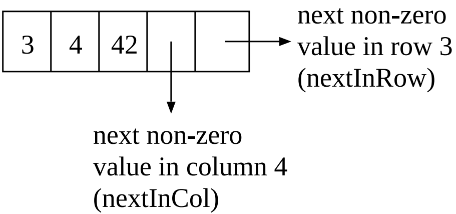
Idea: Only allocate memory for the entries that are non-zero.

Mechanism:

- Store each non-zero entry in a node.
- Use the heap to get the space for non-zero entries.
- Link non-zero entries by row and by column.
- Each node will need to know its row number and column number and (non-zero) value.

An Example Node

For example, row 3 and column 4 (i.e., entry [3,4]) contains 42:



A Node Class

- Our `SparseMatrix` class contains a private `Node` class.
- The `Node` class implements a non-zero entry in a matrix.

```
public class SparseMatrix
{
  private class Node  // A non-zero matrix entry.
  {
    public int row;
    public int col;
    public int value;
    public Node nextInRow;
    public Node nextInCol;
```

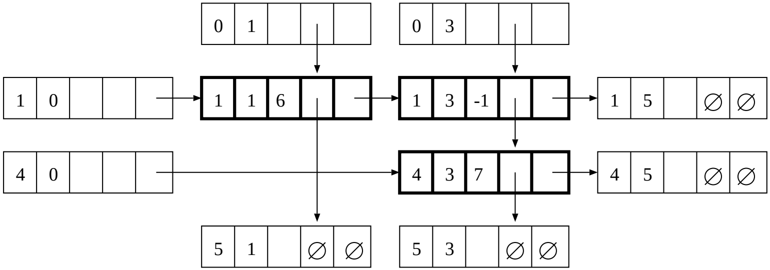
```
    public Node( int R,
                 int C,
                 int newValue,
                 Node rowNext,
                 Node colNext )
    {
      row = R;
      col = C;
      value = newValue;
      nextInRow = rowNext;
      nextInCol = colNext;
    } // end Node constructor

  } // end class Node
...
} // end class SparseMatrix
```

Rows and Columns

- A row or a column is a linked list of non-zero entries.
- We will use plain linked lists (not circular or back-linked) with dummy nodes.
- The dummy header nodes would contain row or column number 0.
- The dummy trailer nodes would contain row number numRows+1 or column number numCols+1.

Example:

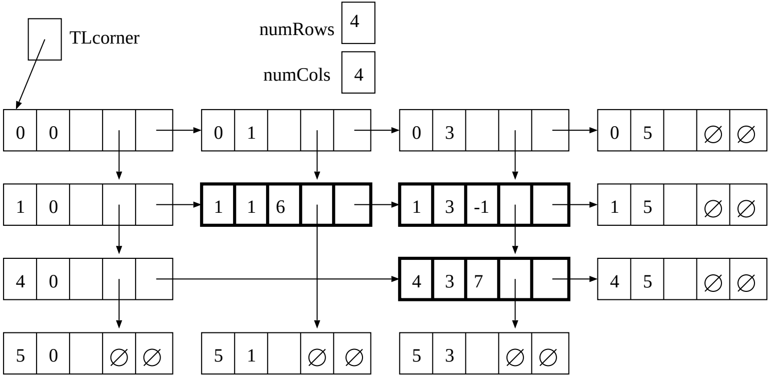
$$\begin{bmatrix} 6 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$



We will need to search for particular rows or columns.
Idea: Link together "row 0", which is a list of all the available columns, and link together "column 0", which is a list of all the available rows.
How about dummy nodes for *those* lists? Good idea.

Example (re-examined):

$$\begin{bmatrix} 6 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$

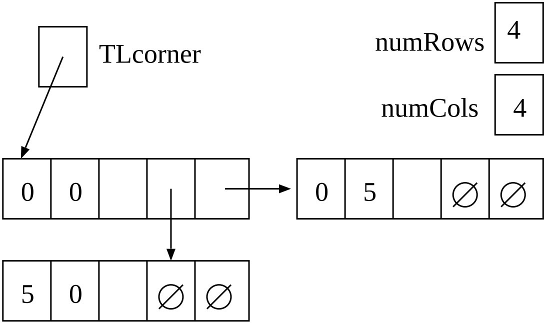


```
public class SparseMatrix
{
  private class Node
  {
    ...
  } // end class Node

  private Node TLcorner; // points to (0,0) node
  private int numRows;   // no. of rows in matrix
  private int numCols;   // no. of columns in matrix
```

```
  public SparseMatrix( int rowSize, int colSize )
   // construct a rowSize x colSize all-zero matrix

  public int getValue( int R, int C )
   // return the value of entry [R,C]

  public void setValue( int R, int C, int newValue )
   // set the value of entry [R,C] to newValue

} // end class SparseMatrix
```

Constructor

The constructor creates an all-zero (empty) matrix of the given size:



```
public SparseMatrix( int rowSize, int colSize )
{
  numRows = rowSize;
  numCols = colSize;
  TLcorner = new Node( 0, 0, 0,
                       new Node( 0, colSize+1, 0,
                                 null, null ),
                       new Node( rowSize+1, 0, 0,
                                 null, null )  );
} // end SparseMatrix constructor
```

Searching For an Entry

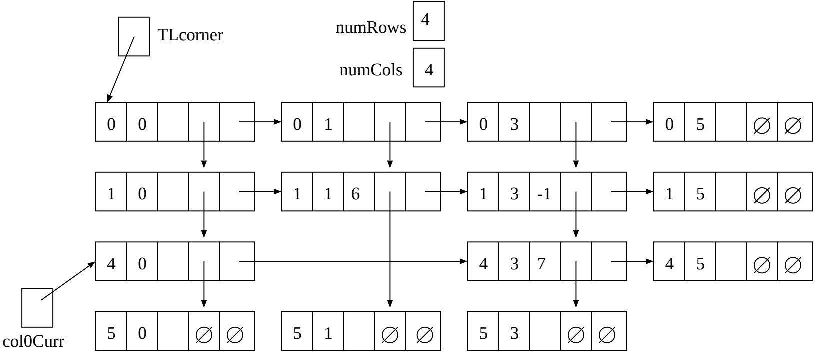
Both `getValue` and `setValue` need to be able to find a particular entry (given the row number, `R`, and column number, `C`, of the entry) — or to notice that there is no such entry.

General Search Strategy

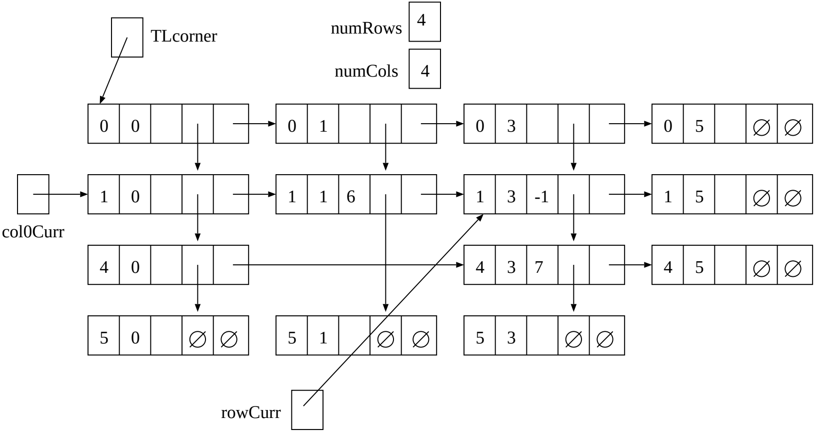
- Search down dummy column 0 looking for the row list for row `R` (or search across dummy row 0 looking for the column list for column `C`).
  If the row list (or column list) doesn't exist, then all entries in that row (or column) are 0.
- Then search along row `R` looking for a node in column `C` (or search down column `C` looking for a node in row `R`).
  If the node you're looking for doesn't exist, then the value of that entry is 0.

Since the entries in a row are sorted by column number (and the entries in a column are sorted by row number), we are always searching in an *ordered* linked list.

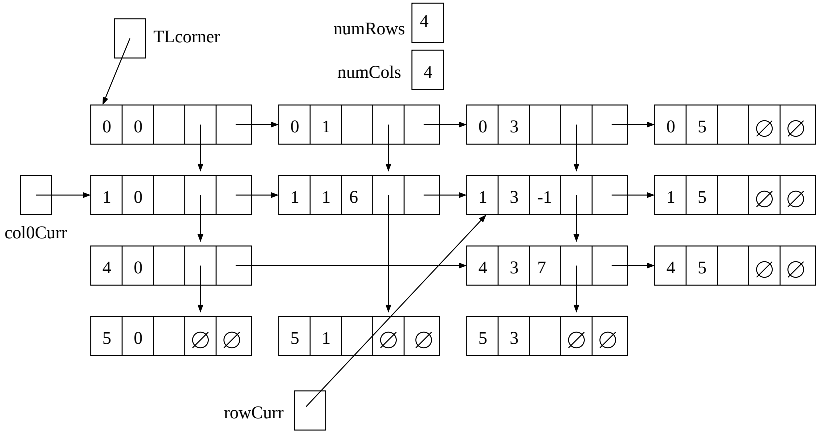
We can see that the value of entry `[3,2]` is zero because there is no list for row 3:



We can see that the value of entry [1,2] is zero because, although there is a list for row 1, the list does not contain a node in column 2:



We can see that the value of entry `[1,3]` is -1 because there is a list for row 1, it contains a node in column 3 and that node contains the value -1.



`getValue( R, C )`

- Returns the value of entry `[R,C]`.
- If we can't find a node for entry `[R,C]`, then the value of entry `[R,C]` is 0.

`getValue(R,C)` uses two other methods:

`findRow(R)`: Returns a pointer to the dummy header node for row `R` (or `null` if there is no list for row `R`).
Searches down dummy column 0, looking for row `R`.

`searchRow(rowTop,C)`: Returns a pointer to the node for entry `[R,C]` or `null` if no such node exists.
Searches along row `R`, starting at the dummy header node for row `R`, which is pointed at by the parameter `rowTop`.

```
public int getValue( int R, int C )
{
  Node theNode;                 // entry [R,C]
  Node rowTop = findRow( R ); // header for row R
  int entryRCvalue = 0;         // value of entry [R,C]

  if (rowTop != null)
  {
    theNode = searchRow( rowTop, C );
    if (theNode != null)
      entryRCvalue = theNode.value;
  } // end if

  return entryRCvalue;

} // end method getValue
```

`findRow(R)`:

- Uses a pointer `col0Curr` to search along column 0, looking for row `R`.
- `col0Curr` starts at the dummy header node for column 0, which is pointed to by `TLcorner`.
- `findRow` can stop searching and return `null` if it reaches a row number greater than `R`.

```
private Node findRow( int R )
{
  Node col0Curr = TLcorner;  // Search column 0.
  Node rowRlist = null;  // Result to return.

  // Move along column 0, looking for row R.
  while (col0Curr.row < R)
    col0Curr = col0Curr.nextInCol;

  // If we found row R, return dummy header of
  // list for row R. Otherwise, return null.
  if (col0Curr.row == R)
    rowRlist = col0Curr;

  return rowRlist;
} // end method findRow
```

`searchRow(rowTop,C)`:

- Uses a pointer `rowCurr` to search along the given row for an entry in column `C`.
- Parameter `rowTop` points to the dummy header node for the row, so we start the search with `rowCurr` pointing at the first real node (the one after the header).

```
private Node searchRow( Node rowTop, int C )
{
  Node rowCurr = rowTop.nextInRow; // Search row.
               // rowTop is the dummy header for row.
  Node entryRC = null; // Result to return.

  // Search row until we're at (or past) column C.
  while (rowCurr.col < C)
    rowCurr = rowCurr.nextInRow;

  if (rowCurr.col == C)
    entryRC = rowCurr;

  return entryRC;
} // end method searchRow
```

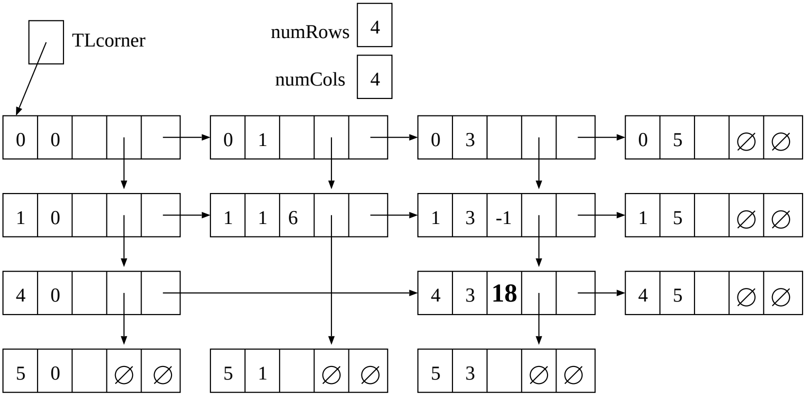
`setValue( R, C, newValue )`

What you have to do may depend on whether:

- `newValue` is 0 or non-zero.
- A node already exists for entry `[R,C]` or not.
- Row `R` contains other non-zero entries or not.
- Column `C` contains other non-zero entries or not.

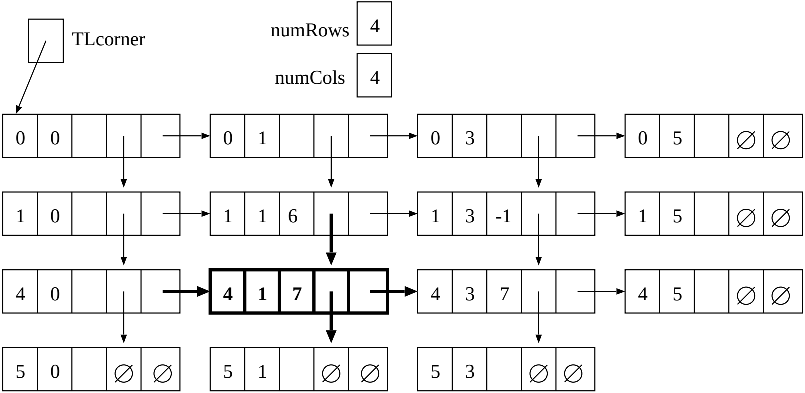
If `newValue` is not 0 and a node already exists for entry `[R,C]`, simply change the value in the existing node.

setValue(4,3,18) causes the following change:



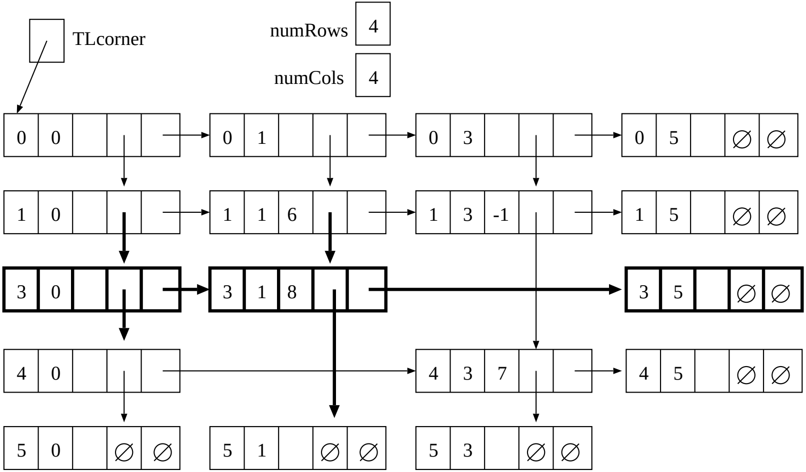
If `newValue` is not 0, but no node exists for entry `[R,C]`, add a new node for entry `[R,C]`. If lists for both row `R` and column `C` already exist, simply link the new node into those lists in the appropriate positions:

setValue(4,1,7) causes the following changes:



If `newValue` is not 0, no node exists for entry `[R,C]`, but no list exists for row `R`, then add a new node for entry `[R,C]` and create a new row list for row `R`.

setValue(3,1,8) causes the following changes:



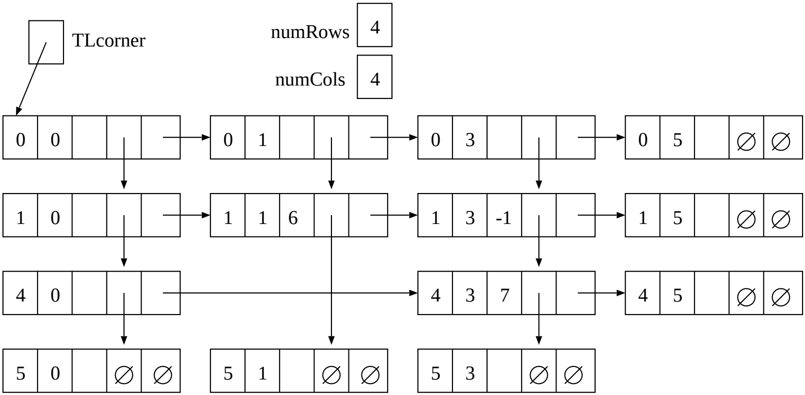
If `newValue` is not 0, you might have to add a new list for column `C`.

You might have to add new lists for both row `R` and column `C`.

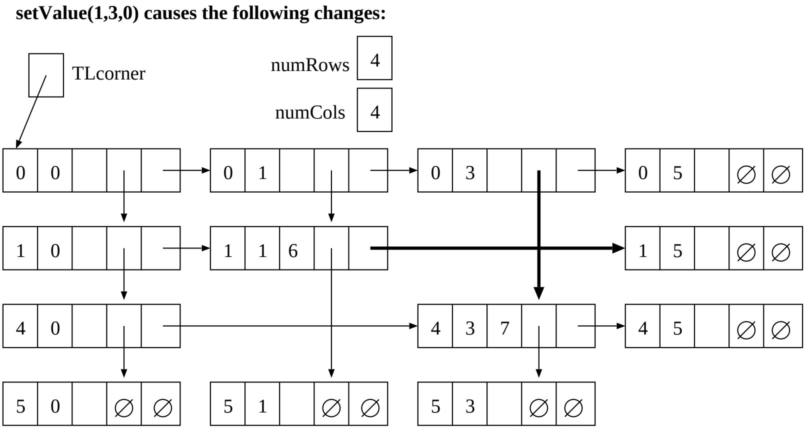
(Pictures not shown.)

If `newValue` is 0 and there is no node for entry `[R,C]`, then do nothing!

setValue(3,2,0) does nothing:

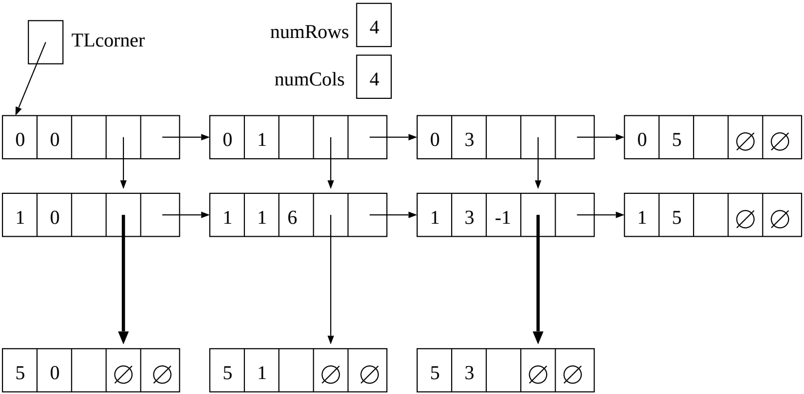


If `newValue` is 0 and there is a node for entry `[R,C]`, then delete that node. If row `R` and column `[C]` both contain other non-zero entries, simply unlink that node from the row list and the column list.



If `newValue` is 0, a node for entry `[R,C]` exists, but row `R` contains no other non-zero entries, then you have to delete the entire row `R`.

setValue(4,3,0) causes the following changes:



If `newValue` is 0 and you delete the only non-zero value in column `C`, then you have to delete the column list for column `C`.
You might have to delete both the list for row `R` and the list for column `C`.
(Pictures omitted.)