CSE221

Lecture 2: Arrays

2021 Fall

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides used in former lectures at UNIST.

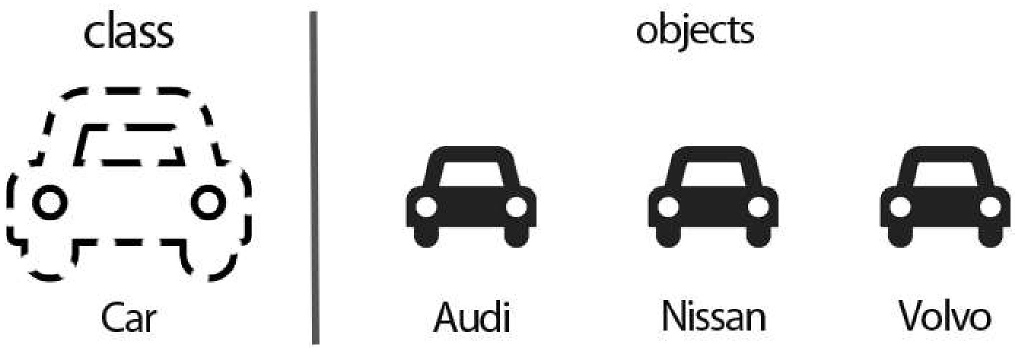
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

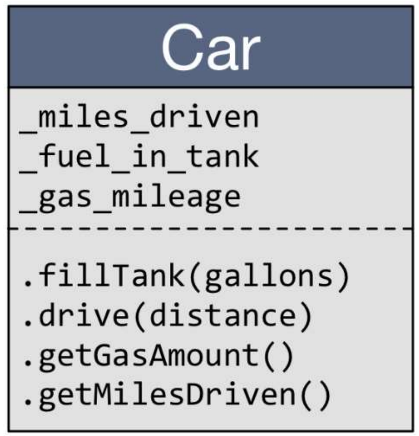
Outline

- Basic concepts
- Arrays
- Examples
  - Sparse matrices
  - Selection sort

Object

- Entity that performs *computations* and has a *local state*
  - –Procedural elements + variables
- An instance of a class





Credits: https://javatutorial.net/java-oop

Object-oriented Design

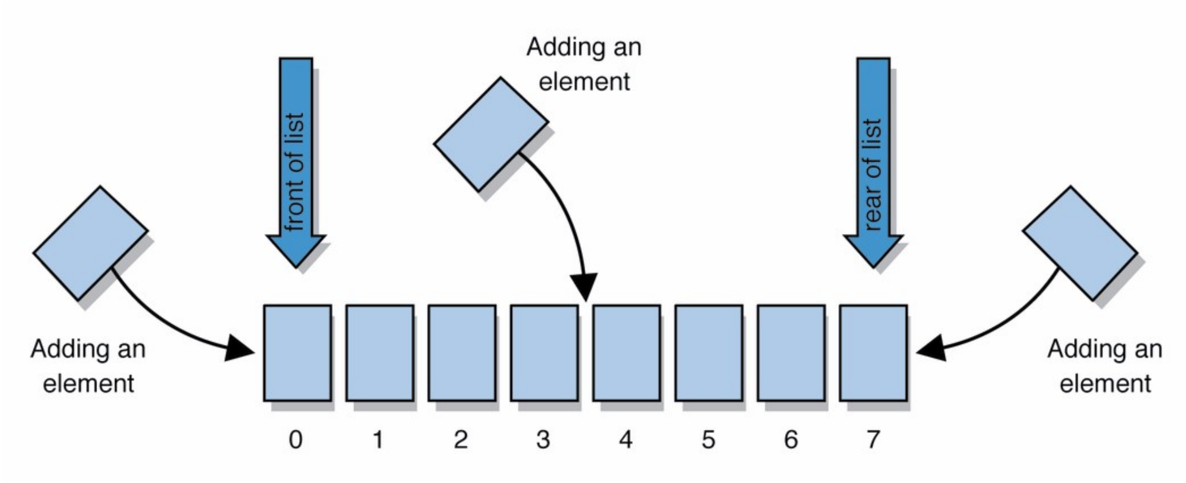
- Object as fundamental building block
  - View software as a set of interacting objects that model entities
  - Use *divide and conquer*
- Benefits
  - Encourage code reusability and flexibility
  - Localize changes: related objects *only*

Data Encapsulation and Abstraction

- Data encapsulation
  - Concealing of the implementation details of a data object from the outside
- Data abstraction
  - Separation between the specification (what) of a data object and its implementation (how)

Abstract Data Type (ADT)

- Definition for expected operations and behavior
- Example: list
  - A collection storing an ordered sequence of elements
  - Operations:



Data Structure and ADT

- Data structure
  - How we organize/store data points
  - Determines how each operation would behave
- One ADT can be implemented in many ways
- One data structure can be used for multiple ADTs

ADT First, Then Data Structure

- Wrong way:
  - I will represent my data A as an array
- Right way:
  - I will be frequently accessing a random data point of my data A, so I need operation []
  - I will therefore use an array

Benefits of Abstraction and Encapsulation

- Simplification of software development
  - Enables parallel development of subtasks
- Testing and debugging
  - Easier to identify problems by testing subtasks
- Reusability
  - Easier to take out a part and use elsewhere
- Modification of data type
  - Implementation can be modified without changing i nterfaces

Outline

- Basic concepts
- Arrays
- Examples
  - –Sparse matrices
  - –Selection sort

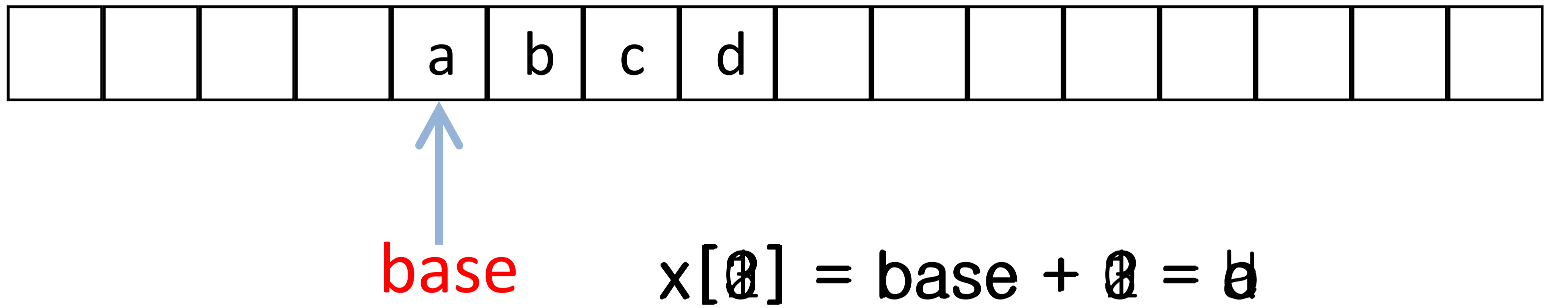
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Arrays in C/C++

- Continuous memory space with the same data type
- `type name [elements];`
- Example:
  - int a[100];
  - float b[10][20];
  - mytype c[5]; // user-defined data type
- Each element accessed by index

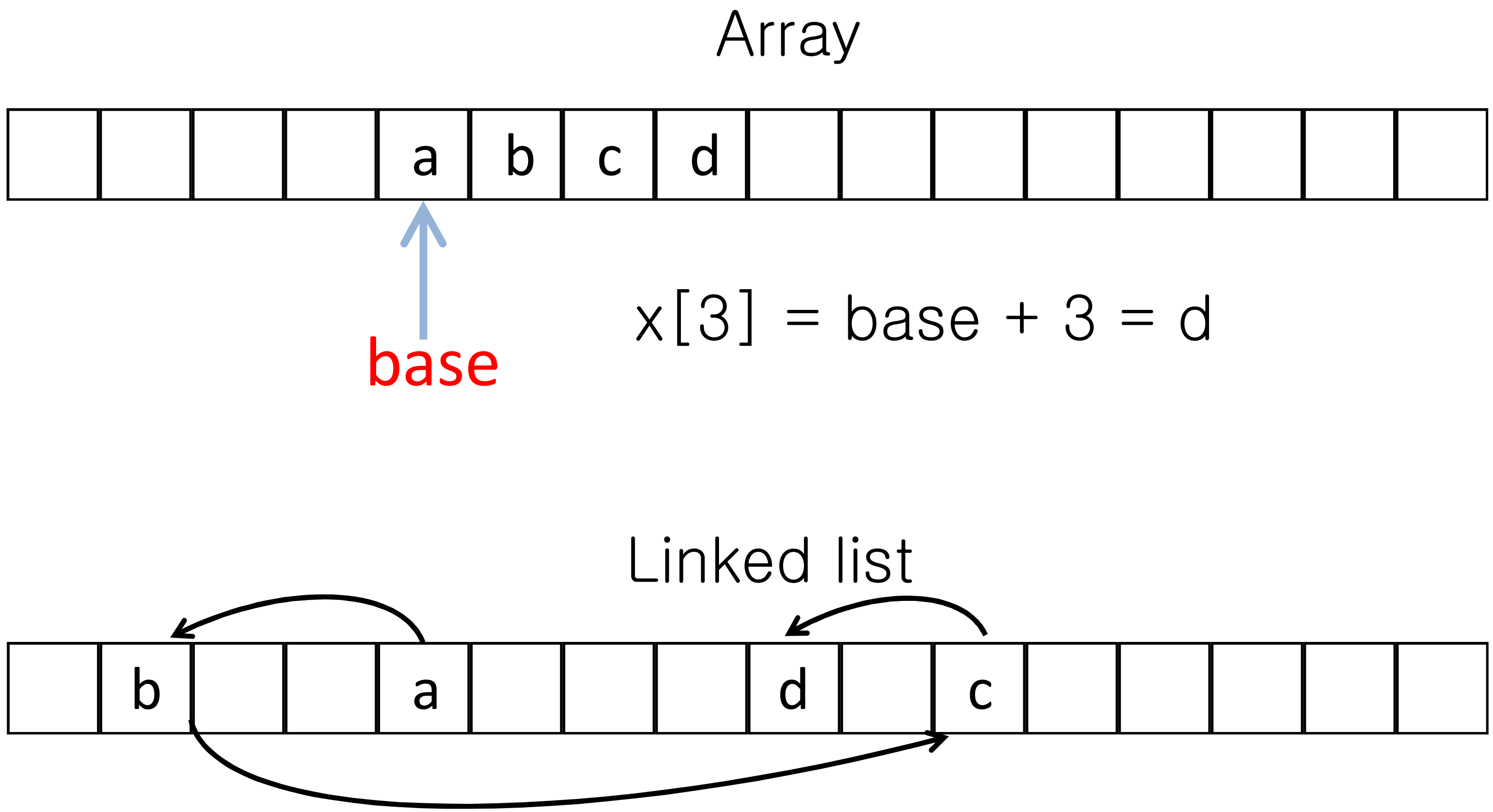
1D Array Representation In C

1D Memory Space



- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations
- location(x[i]) = base + i

Array v.s. Pointer-based Linked List



Array v.s. Pointer-based Linked List

- Arrays
  - Good for random and sequential access
    - Indexing
    - Modern architecture reads data at a chunk
  - Good when the size is quite fixed
- Linked list
  - Good for frequent inserting and deleting
  - Resizing is easier

2D Arrays

- The elements of a 2-dimensional array x declared as:
- int x[3][4];
- may be shown as a table

x[0][0]	x[0][1]	x[0][2]	x[0][3]
x[1][0]	x[1][1]	x[1][2]	x[1][3]
x[2][0]	x[2][1]	x[2][2]	x[2][3]

Rows of a 2D Array

x[0][0] x[0][1] x[0][2] x[0][3] → row 0

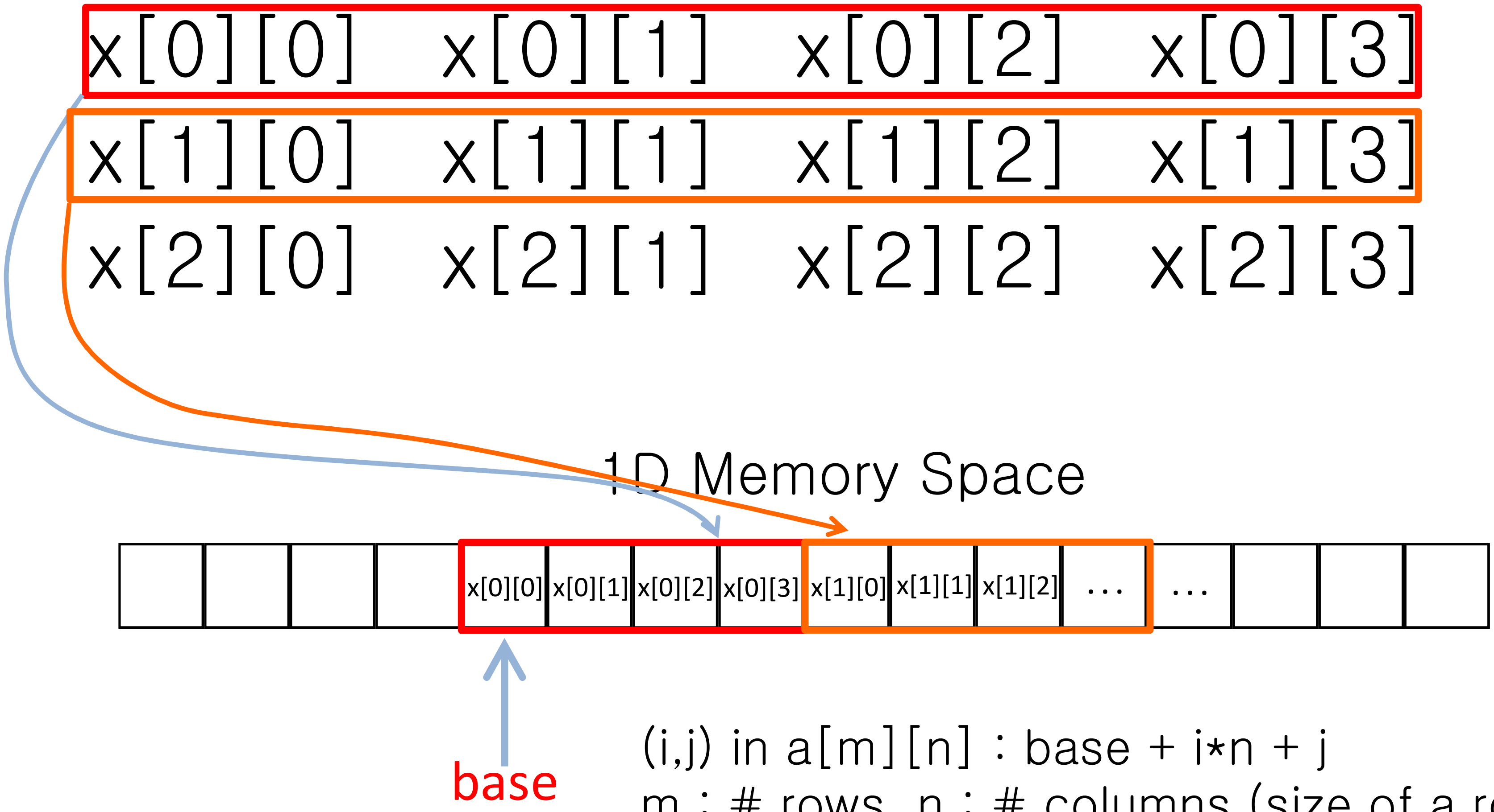
x[1][0] x[1][1] x[1][2] x[1][3] → row 1

x[2][0] x[2][1] x[2][2] x[2][3] → row 2

Columns of a 2D Array

x[0][0]	x[0][1]	x[0][2]	x[0][3]
x[1][0]	x[1][1]	x[1][2]	x[1][3]
x[2][0]	x[2][1]	x[2][2]	x[2][3]
col 0	col 1	col 2	col 3

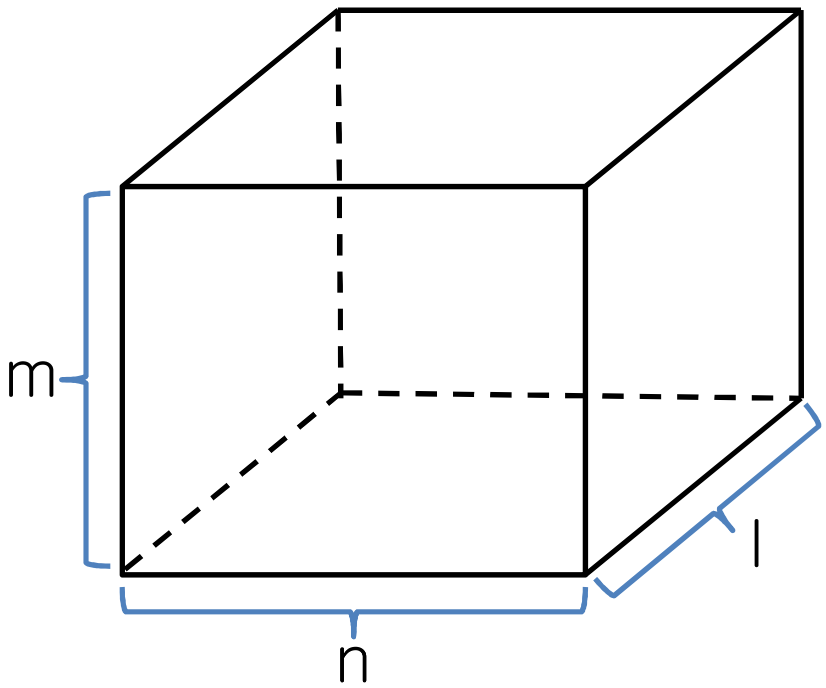
1D Indexing of 2D Array



(i,j) in a[m][n] : base + i*n + j
m : # rows, n : # columns (size of a row)
i : row index, j : column index

1D Indexing of 3D Array

- (i,j,k) in x[l][m][n] : base + i*m*n + j*n + k



ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Array Abstract Data Type

- More general than array in C/C++
  - Contiguous memory location is more of a implementation issue
- Properties:
  - Mapping between index and value
    - Do not need to be implemented with pure array!
  - Operators to retrieve and store value

```
class GeneralArray {
// A set of pairs <index, value> where for each
// value of index in index set, there is a value of
// type float
public:
      // Constructor.
      // j : dimension, list : range of each dimension.
      // initValue : initial value
      GeneralArray(int j,
                   RangeList list,
                   float initValue);

      // Return the float associate with i if i is in
      // the index set. Otherwise throw an exception
      float Retrieve(index i);

      // Replace the old value associate with i if i is
      // in the index set. Otherwise throw an exception
      void Store(index i, float x)
}
```

Outline

- Recap OOP: programming concepts
- Arrays
- Examples
  - Sparse matrices
  - Selection sort

Matrices

- A[m][n]
  - m x n matrix A
  - m : # of rows
  - n : # of column
  - m*n : # of elements
- Accessing an element at i-th row and j-th col
  - A[i][j]
- 2D array is often used to represent matrices → Dense matrix

Sparse Matrices

- Mostly zero

$$
\begin{array}{c c}
 & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} &
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
$$

(a) Dense Matrix

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

(b) Sparse Matrix

Sparse Matrix Representation

- Array of triples <row, col, value> for nonzero elements
- # elements to store (space requirement) for nxn matrix
  - –Dense : n^2
  - –Sparse diagonal matrix : 3*n

Meaningful when n is very large!

Tridiagonal Matrix

$$f_n = \begin{vmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{vmatrix}.$$

Row Major vs. Column Major

- 2D->1D layout
- Row major
  - Store each row in contiguous memory
- Column major
  - Store each column in contiguous memory

Sparse Matrix

- Array of triples <row, col, value> for nonzero elements

```
class SparseMatrix;
    class MatrixTerm {
    friend class SparseMatrix;
    private:
        int row, col, value;
    };

private:
    int rows, cols, terms, capacity;
    MatrixTerm * smArray;
```

Example

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 15 & 0 & 0 & 22 & 0 & -15 \\
1 & 0 & 11 & 3 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & -6 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 91 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
$$

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

Row-major ordering

Space Cost Analysis

1 integer

$$\begin{array}{c|c}
 & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\
\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}$$

3 integers
(3x more space)

	row	col	val
[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

space saving = 12
(36/24 integers in dense/sparse matrix)

- Sparse representation is more effective when there are many empty elements

Sparse Matrix Transpose

- Exchange (i,j) to (j,i)
- Pseudo code:
  - for each row i, store (i,j) to (j,i)
  - Problem

    (0, 0, 15) -> (0, 0, 15)

    (0, 3, 22) -> (3, 0, 22)

    (0, 5, -15) -> (5, 0, -15)

    (1, 1, 11) -> (1, 1, 11) **X** Violate row-major order

Sparse Matrix Transpose

- New approach
  - Use column major order traversal
    - After transpose, matrix is row-major order

```
for (all element in column j)
  store (i,j,value) of the original matrix as (j,i,value) of transposed matrix
```

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]			
	[3]			
	[4]			
	[5]			
	[6]			
	[7]			

for Column 0

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]			
	[4]			
	[5]			
	[6]			
	[7]			

for Column 1

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]			
	[6]			
	[7]			

for Column 2

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]	3	0	22
	[6]	3	2	-6
	[7]			

for Column 3

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]	3	0	22
	[6]	3	2	-6
	[7]			

for Column 4 : No Match

Example

		row	col	val
smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]	3	0	22
	[6]	3	2	-6
	[7]	5	0	-15

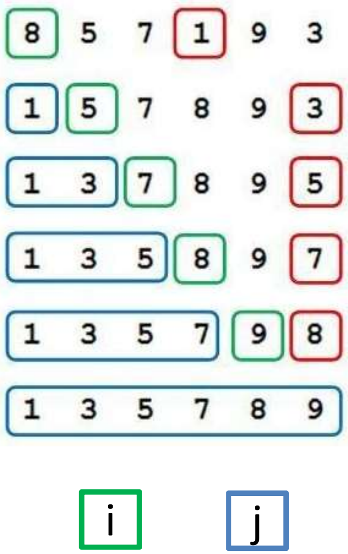
for Column 5

```
SparseMatrix SparseMatrix::Transpose(){
    SparseMatrix b;             // b : output (transposed matrix)
    b.Rows = Cols;              // b rows = a cols
    b.Cols = Rows;              // b cols = a rows
    b.Terms = Terms;            // # of elements is same

    if (Terms > 0){             // non empty matrix
        int CurrentB = 0;
        for (int c = 0; c < Cols; c++)          // per each column
            for (int i = 0; i < Terms; i++)     // for all nonzero elements in sparse
                if (smArray[i].col ==c) {       // matrix
                    b.smArray[CurrentB].row = c;
                    b.smArray[CurrentB].col = smArray[i].row;
                    b.smArray[CurrentB].value = smArray[i].value;
                    CurrentB++;
                }
    }
    return b;
}
```

Selection Sort

- *From those integers that are currently unsorted, find the smallest and place it next in the sorted list*




```
for (int i = 0; i < n; i++) [
   examine a[i] to a[n-1] and suppose the smallest is at a[j];
   interchange a[i] and a[j];
}
```

This is algorithm!

Selection Sort

```
void sort(int *a, const int n)
// sort n integers a[0] to a[n-1] into nondecreasing order.
{
    for (int i = 0; i < n; i++)
    {
        int j = i;
        // find smallest integer in a[i] and a[n-1]
        for (int k = i + 1; k < n; k++)
            if (a[k] < a[j]) j = k;
        // swap
        int temp = a[i]; a[i] = a[j]; a[j] = temp;
     }
 }
```

This is C++ program!

Questions?