Lecture 2

- Array
- Dynamic array
- Matrix

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Many slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

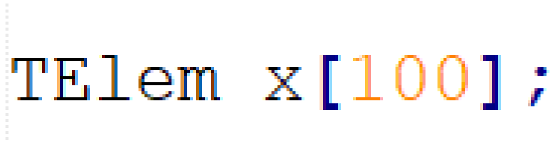
Arrays

Specific:

- elements occupy a contiguous memory block
- any element of the array can be accessed "directly": in **Θ(1)**

  the address of the element can be computed simply

In C/C++:



```
TElem x[100];
```

x: TElem[100]
(in pseudocode, our conventions)

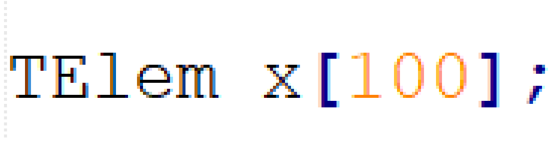
- The first element is at position 0
- Address of i-th element : address of array + i * size_of TElem

What would be the formula for computing the address of the element x[i] if the first position would be 1?

Arrays

- An array is one of the simplest and most basic data structures.
- An array can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block.
- Arrays are often used as a basis for other (more complex) data structures.

Array in C / C++

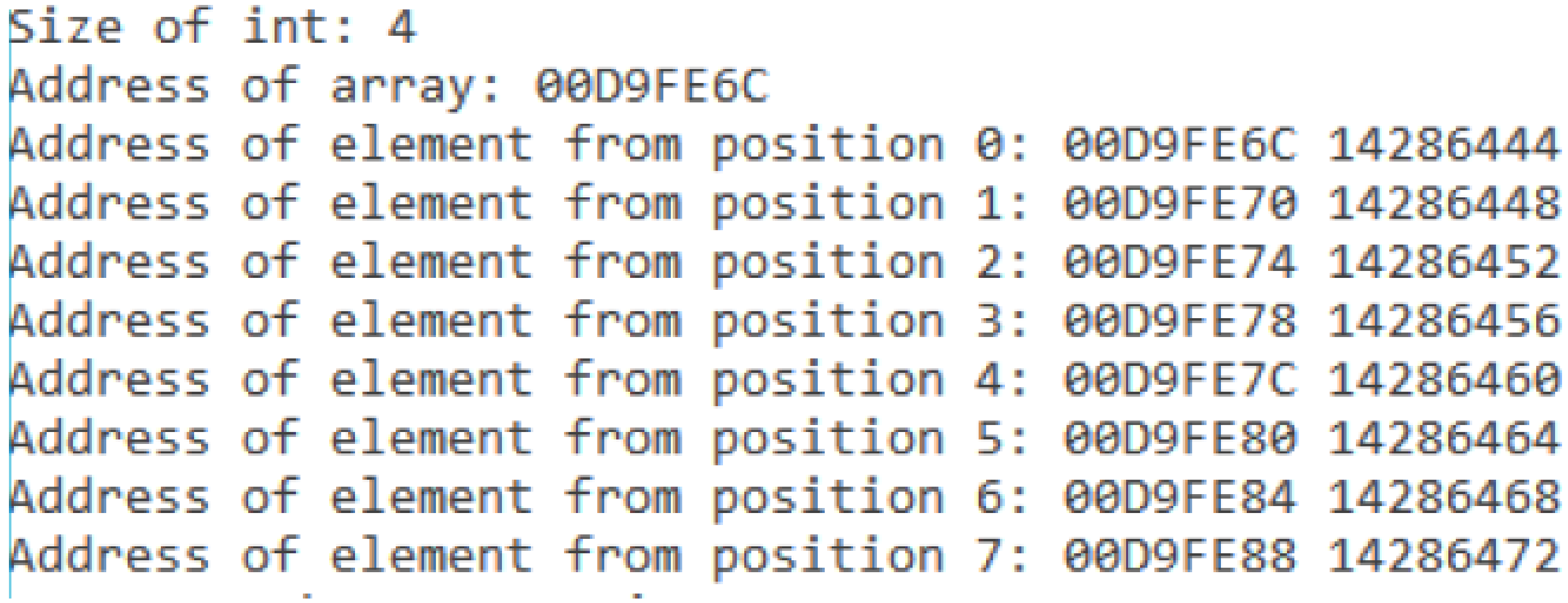


```
TElem x[100];
```

- When a new array is created we have to specify two things:
  - The type of the elements in the array
  - The maximum number of elements that can be stored in the array (capacity of the array)
- The memory occupied by the array will be the capacity times the size of one element.
- The array itself is memorized by the address of the first element.

Array in C / C++ . Example

An array of integer values (integer values occupy 4 bytes)



```
Size of int: 4
Address of array: 00D9FE6C
Address of element from position 0: 00D9FE6C 14286444
Address of element from position 1: 00D9FE70 14286448
Address of element from position 2: 00D9FE74 14286452
Address of element from position 3: 00D9FE78 14286456
Address of element from position 4: 00D9FE7C 14286460
Address of element from position 5: 00D9FE80 14286464
Address of element from position 6: 00D9FE84 14286468
Address of element from position 7: 00D9FE88 14286472
```

- Can you guess the address of the element from position 8?

Arrays

In C/C++:

```
TElem x[100];
```

x: TElem[100]

- This is a **static structure**: once the capacity of the array is specified, you cannot add or delete slots from it (you can add and delete elements from the slots, but the number of slots, the capacity, remains the same)

Disadvantage:

- we need to know/estimate from the beginning the number of elements:
  - if the capacity is too small: we cannot store every element we want to
  - If the capacity is too big: we waste memory

Solution ?

Use **dynamic** arrays !

Dynamic array

Specific:

- can grow or shrink (at execution time)
- and are still arrays (use contiguous memory locations)

How can we work with them?

- Use libraries

e.g.: C++ , STL library, vector

Please do not use it for DSA labs!

- Implement them by using pointers (and pointer operations)

C++, operations with pointers
e.g.:

```
TElem *x;
x = new TElem[cap] ;
delete[] ;
```

Dynamic array: DS or ADT ?

Dynamic Array is a data structure:

- It is strongly related to: how data is actually stored in the computer (in a single contiguous memory block) and how it can be accessed and processed

In C / C++ :

```
int cap;
int nrElem;
TElem *elems;
```

(Maybe encapsulated in a class)

In pseudocode:

```
DynamicArray:
    cap: Integer
    nrElem: Integer
    elems: TElem[]
```

(The default DS for us)

In most programming languages

it exists as a separate **container** as well.

We can see it as ADT (domain + operations).

ADT DynamicArray

<u>For specification please see:</u>

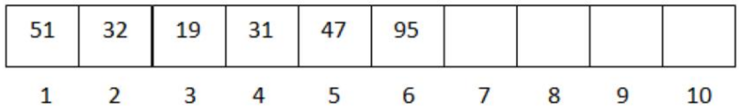
Lecture02 - DynamicArray as ADT.pdf

Operations:	(when implemented over a dynamic array)
• init	**Θ(1)**
• destroy	**Θ(1)**
• size	**Θ(1)**
• getElement	**Θ(1)**
• setElement	**Θ(1)**
• addToPosition	O(n)
• deleteFromPosition	O(n)
• deleteFromEnd	**Θ(1)**
• addToEnd	**Θ(1)** amortized
• iterator	**Θ(1)**

Indexed access for a lot of operations

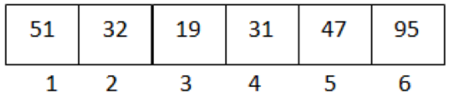
DynamicArray: addToEnd

e.g.:



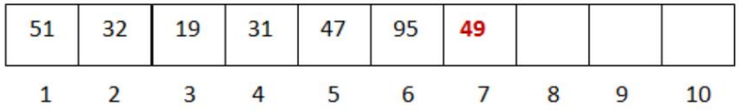
- capacity (cap): 10
- nrElem: 6

- Add the element 49 to the end of the dynamic array

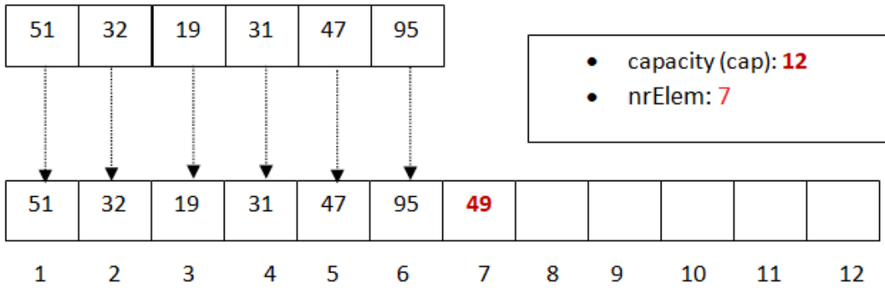


- capacity (cap): 6
- nrElem: 6

- **Add the element 49 to the end of the dynamic array**



- capacity (cap): 10
- nrElem: 7



- capacity (cap): **12**
- nrElem: 7

DynamicArray: addToEnd

Consider the next subalgorithm:

```
subalgorithm addToEnd (da, e) is:
    if da.nrElem = da.cap then
    //the dynamic array is full. We need to resize it
        da.cap ← da.cap * 2
        newElems ← @ an array with da.cap empty slots
        //we need to copy existing elements into newElems
        for index ← 1, da.nrElem execute
            newElems[index] ← da.elems[index]
        end-for
        //we need to replace the old element array with the new one
        //depending on the prog. lang., we may need to free the old elems array
        da.elems ← newElems
    end-if
    //now we certainly have space for the element e
    da.nrElem ← da.nrElem + 1
    da.elems[da.nrElem] ← e
end-subalgorithm
```

Use @ … like here for array allocation specified in pseudocode

After a resize operation the elements of the Dynamic Array will still occupy a contiguous memory zone, but it will be a different one than before.

What is the complexity of addToEnd?

DynamicArray, addToEnd: amortized analysis

- In amortized time complexity analysis we consider a sequence of operations and compute the average time for these operations.
- Consider c_i the cost (number of instructions) for the ith call to addToEnd

$$c_i = \begin{cases} i, & \text{if i-1 is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- Cost of n operations is:

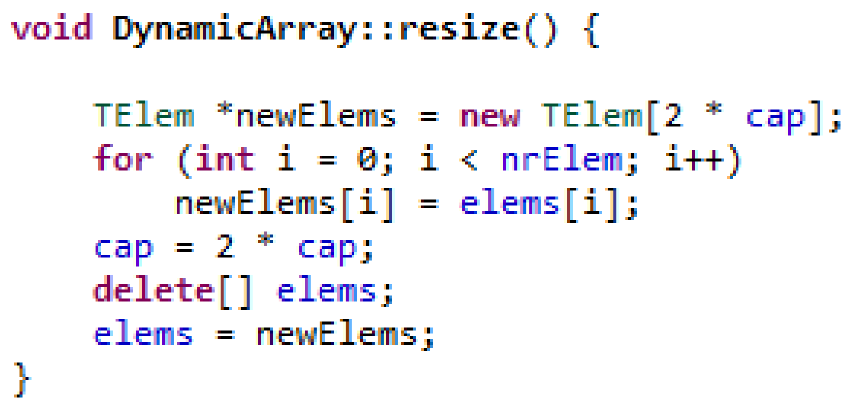
$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{[log_2 n]} 2^j < n + 2n = 3n$$

Since the total cost of n operations is 3n, we can say that the cost of one operation is 3, which is constant.
=> the amortized complexity is **Θ(1)**

DynamicArray: addToEnd & resize

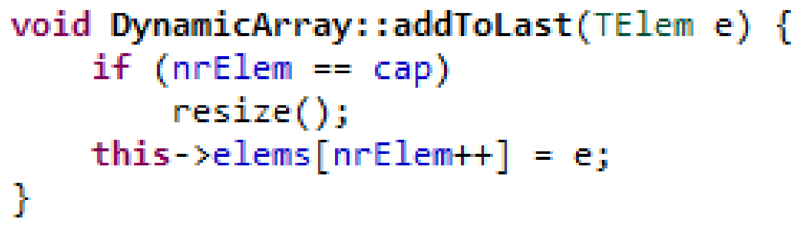
- All operations that allows adding elements in a container will have to consider a resize of the dynamic structure
- => implement a subalgorithm **resize(da)**

e.g.: C++



```
void DynamicArray::resize() {

    TElem *newElems = new TElem[2 * cap];
    for (int i = 0; i < nrElem; i++)
        newElems[i] = elems[i];
    cap = 2 * cap;
    delete[] elems;
    elems = newElems;
}
```



```
void DynamicArray::addToLast(TElem e) {
    if (nrElem == cap)
        resize();
    this->elems[nrElem++] = e;
}
```

Describe same alg. in pseudocode !

What is the complexity of addToEnd?

DynamicArray: resize

- While it is not mandatory to double the capacity, it is important to define the new capacity as **a product** of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK).

How do dynamic arrays in other programming languages grow at resize?

- Microsoft Visual C++ - multiply by 1.5

  (initially 1, then 2, 3, 4, 6, 9, 13, etc.)
- Java - multiply by 1.5 (initially 10, then 15, 22, 33, etc.)
- C# - multiply by 2 (initially 0, 4, 8, 16, 32, etc.)

DynamicArray: expansion and contraction

Consider DELETE operation:
it is enough to remove the specified item

- It is often desirable to contract the table, to reduce the wasted space
- Table contraction is analogous to table expansion
  allocate a new, smaller table and then copy the items

A natural strategy (?!)

- Expansion: double the table capacity when an item is inserted into a full table
- Contraction: halve the capacity when a deletion would cause the table to become less than half full.

How empty should the array become before resize? Which of
the following two strategies do you think is better? Why?

- Wait until the table is only half full (da.nrElem ≈ da.cap/2) and resize it to the half of its capacity
- Wait until the table is only a quarter full (da.nrElem ≈ da.cap/4) and resize it to the half of its capacity

Matrix

<u>Specific</u>:

- two-dimensional array.
- each element has a unique position, determined by two indexes: its line and column.

<u>e.g.</u>: **C / C++**

```
TElem x[3][3];
```

- usually a sequential representation is used for a Matrix

  (we memorize all the lines one after the other in a consecutive memory block).
- If this sequential representation is used, for a matrix with N lines and M columns, the element from position (i ; j) can be found at the memory address:

  address_of_element_from_position (i, j) =

  address_of_the_matrix + (i * M + j)* *size_of_an_element*

  The above formula works for 0-based indexing, but can be adapted to 1-based indexing as well.

Imagine a DT Matrix. What operations should we have for a Matrix?

ADT Matrix

Please see:
Lecture02_ADTMatrix (assumes 1-based indexing)
ADTMatrix interface (Lab) (assumes 0-based indexing)

$\mathcal{D}_{MAT}$ = { mat | mat is a matrix with elements of the type TElem}

Operations:

- init(mat, nrL, nrC)
- nrLines(mat)
- nrCols(mat)
- element(mat, i, j)
- modify(mat, i, j, val)

There is no operation to add an element or to remove an element. In the interface we only have the modify operation which changes a value from a position.

Other possible operations:

- get the first position of a given element
- create an iterator that goes through the elements by columns
- create an iterator the goes through the elements by lines

Sparse Matrix

If the matrix contains many values of 0 (or 0_{TElem}), we have a sparse matrix, where it is more (space) efficient to memorize only the elements that are different from 0.

- We can memorize (line, column, value) triples, where value is different from 0 (or 0TElem).

  For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column).

- compressed sparse line representation /

  compressed sparse column representation

Sparse Matrix , Triples

e.g.

- Number of lines: 6
- Number of columns: 8

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

Triples: (line, column, value)

- can be stored in a dynamic array

(or in other data structures)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

Sparse Matrix , Triples

e.g.:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	1	1	1	2	2	2	2	3	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

- Modify the value from position (6, 4) to 1 ? ☺

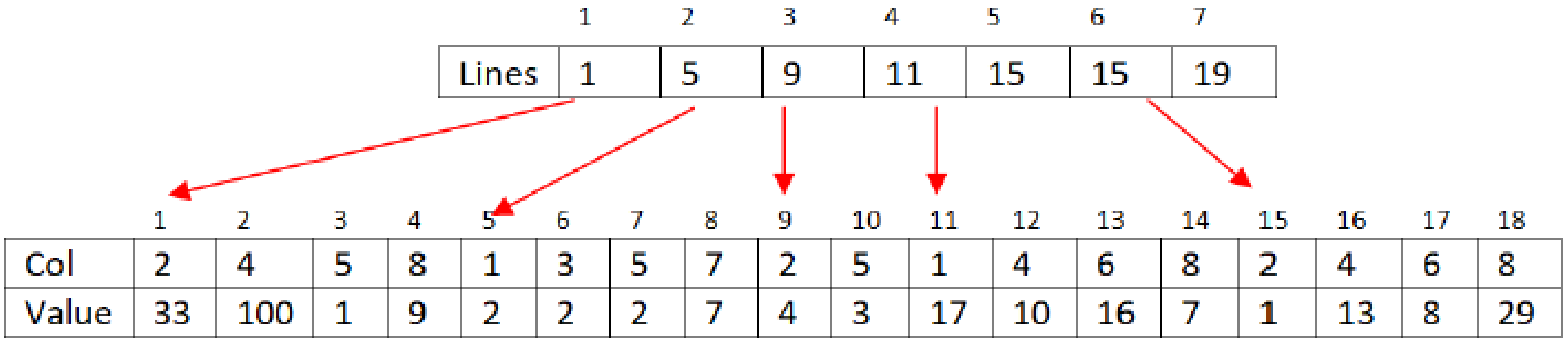
Sparse Matrix , compressed sparse line

e.g.

- Number of lines: 6
- Number of columns: 8

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

Compressed sparse line



	1	2	3	4	5	6	7
Lines	1	5	9	11	15	15	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

Sparse Matrix , compressed sparse line

- Use Col and Value arrays.

- For the lines, have an array of number of lines + 1
  in which at position i we have the position from the Col
  array where the sequence of elements from line i begins.

- Thus, elements from line i are in the Col and Value arrays between the positions [Line[i], Line[i+1]).

- In order for this representation to work, in the Col and Value arrays the elements have to be stored by rows (first elements of the first row, then elements of second row, etc.)

Sparse Matrix , compressed sparse line

e.g.:

	1	2	3	4	5	6	7
Lines	1	5	9	11	15	15	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

	1	2	3	4	5	6	7
Lines	1	4	8	10	14	14	18

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

Sparse Matrix , compressed sparse line

e.g.:

	1	2	3	4	5	6	7
Lines	1	4	8	10	14	14	18

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

	1	2	3	4	5	6	7
Lines	1	4	8	11	15	15	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 6) to 0 ? ☺

Sparse Matrix , compressed sparse column

Representation:

- an array with nrColumns + 1 elements, in which at position i we have the position from the Lines array where the sequence of elements from column i begins.
- two arrays Lines and Values for the non-zero elements, in which first the elements of the first column are stored, than elements from the second column, etc.

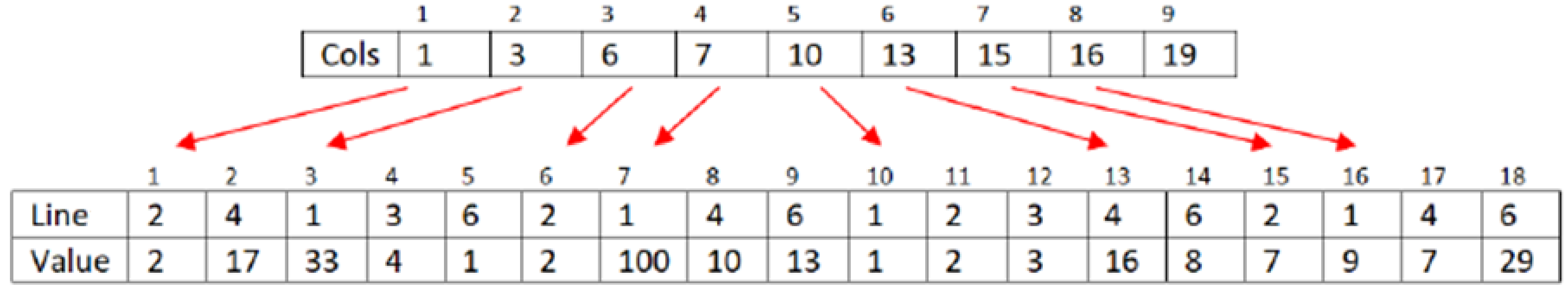
Elements from column i are in the Lines and Value arrays between the positions [Col[i], Col[i+1]).

e.g. :

Number of lines: 6

Number of columns: 8

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29



	1	2	3	4	5	6	7	8	9
Cols	1	3	6	7	10	13	15	16	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line	2	4	1	3	6	2	1	4	6	1	2	3	4	6	2	1	4	6
Value	2	17	33	4	1	2	100	10	13	1	2	3	16	8	7	9	7	29

Sparse Matrix , ex.

For the next sparse matrix,

describe (draw) the representation by using:

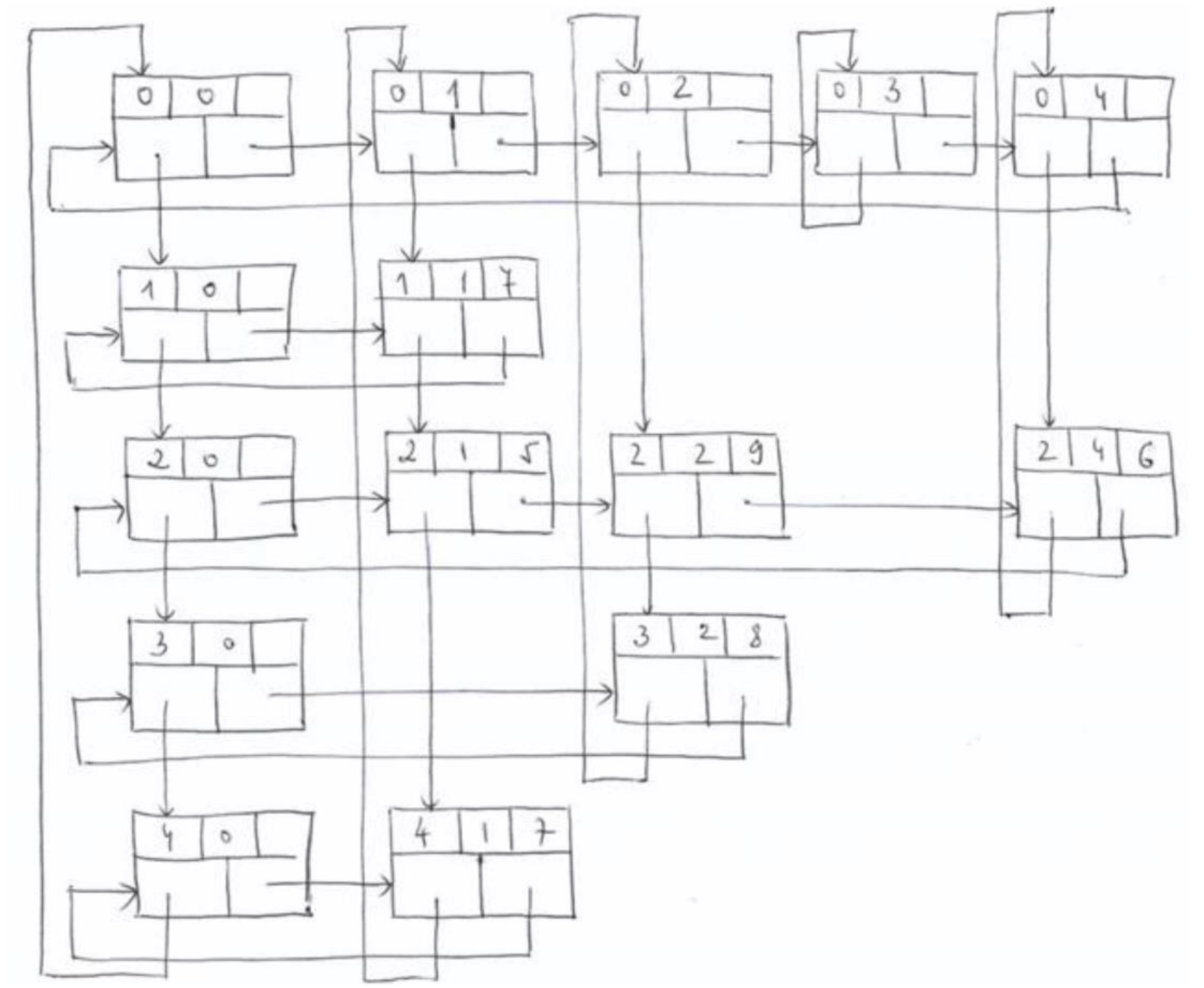
- triplets
- compressed sparse row
- compressed sparse column

7	0	0	0
5	9	0	6
0	8	0	0
7	0	0	0

Sparse Matrix , linked repr.

Interconnected circular list **e.g.**

- Many variants



0 0
0 1
0 2
0 3
0 4
1 0
1 1 7
2 0
2 1 5
2 2 9
2 4 6
3 0
3 2 8
4 0
4 1 7