Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत



**DS221** | 4 Sep – 16 Oct, 2016

# Data Structures, Algorithms & Data Science Platforms

Yogesh Simmhan

simmhan@cds.iisc.ac.in



CDS
The Department of Computational and Data Science

# What we will cover

- **Data Structures & Algos: 8 lectures (Sep 4-27)**
  - Refresher of data structure basics
  - Some "advanced" topics on trees, graphs, concurrent structures
  - Algorithmic analysis and design patterns
  - Some programming tutorials (C++). But students must learn independently.
  - 1 programming assignment, **30 Sep** **[15 points]**
- **Data Science Platforms: 3 lectures (Oct 4-11)**
  - Introduction to Cloud computing, Big Data platforms
  - Apache Spark, tutorial on PySpark
  - 1 short programming assignment, **25 Oct** **[5 points]**
- **Mid-term exam, 16 Oct [10 points]**

# Class Resources

- Website
  - Schedule, Lectures, Assignments, Additional Reading
  - http://cds.iisc.ac.in/courses/ds221/
- Textbook
  - Data Structures, Algorithms, and Applications in C++, 2nd Edition, Sartaj Sahni*,**
    - http://www.cise.ufl.edu/~sahni/dsaac/
- Other resources
  - The C++ Programming Language, 3rd Edition, Bjarne Stroustrup
  - THE ART OF COMPUTER PROGRAMMING (Volume 1 / Fundamental Algorithms), Donald Knuth
  - Introduction to Algorithms, Cormen, Leiserson, Rivest and Stein
  - www.geeksforgeeks.org/data-structures/

*http://www.tatabookhouse.com/data-structures-algorithms-and-applications-in-c-plus-plus--9788173715228?ver=1519259641
**http://www.flipkart.com/data-structures-algorithms-applications-c-english-2nd/p/itmeyf6jvka3kzdu

# Ethics Guidelines

- Students must uphold IISc's Code of Conduct.
  - *Review them!* Failure to follow them **<u>will</u>** lead to sanctions and penalties: reduced or failing grade … **<u>Zero Tolerance!</u>**
- Learning takes place both within and outside the class
  - More outside than inside ☺
- Discussions between students and reference to online material is <u>highly encouraged</u>
- However, you must form your own ideas and **<u>complete problems and assignments by yourself</u>**.
- All works submitted by the student as part of their academic assessment must be their own!

# L1: Introduction

2018-09-04

# Concepts

- **Algorithm**: Outline, the essence of a computational procedure, with step-by-step instructions
- **Program**: An implementations of an algorithm in some programming language
- **Data structure**: Organization of data needed to solve the problem (array, list, hashmap)
- **Algorithmic Analysis:** The expected behaviour of the algorithm you have designed, *before you run it*
- **Empirical Analysis:** The behaviour of the program that implements the algorithm, *by running it*

Why not just run it and see how it behaves?

# Limitation of Empirical Analysis

- Need to implement the algorithm
  - Time consuming
- Cannot exhaust all possible inputs
  - Experiments can be done only on a limited to set of inputs
  - May not be indicative of running time for other inputs
- Harder to compare two algorithms
  - Same hardware/environments needs to be used

# How do we design an algorithm?

- Intuition
- Mixture of techniques, design patterns
- Experience (body of knowledge)
- Data structures, analysis

# How do we implement a program?

- Preferred High Level Language, e.g. C++, Java, Python
- Map algorithm to language, retaining properties
- Use native data structures, libraries

Then why learn about basic data structures?

# Algorithm, Data Structure & Language are interconnected

- Algorithms based on specific data structures, their behavior
- Algorithms are limited to the features of the programming language
  - Procedural, Functional, Object oriented, distributed
- Data structures may/may not be natively implemented in language
  - Java Collections, C++ STL, NumPy

# Basic Data Structures

Lists

# Collections of data

- Data Structures to store collections of data items of same type
  - Items also called `elements, instances, values`…depending on context
- Primitive types can be boolean, byte, integer, etc.
- Complex types can be user or system defined objects, e.g., node, contact, vertex
- Operations on the collection
  - Standard operations to create, modify, access elements
- Properties of the collection
  - **Invariants that must be maintained**, irrespective of operations performed
- **Challenge**: Understand how to pick the right data structure for your application!

# Collections of data

- Different implementations for same abstract collection data type
  - All offer **same** operations and invariant guarantees
  - Differ in performance, space/time complexity
- **Challenge**: Understand how to pick the right implementation!

**Try yourself!**

- Learn **templates/generics**. In many collections, the item type does not matter for invariants and operations, and can be replaced by a place-holder type "T".
- Learn C++ **Standard template library (STL)**. Read up examples of abstract collections and their implementations.

# Linear List (abstract data type)

- Properties
  - **Ordered** list of items…precedes, succeeds; first, last
  - **Index** for each item…lookup or address item by index value
  - **Well-defined size** for the list at a point in time…can be empty, size may vary with operations performed
  - Items of **same type** present in the list

*Type = int, Size = 7*

| *Index* | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|---|
| *Item* | 36 | 5 | 75 | 11 | 7 | 19 | -1 |

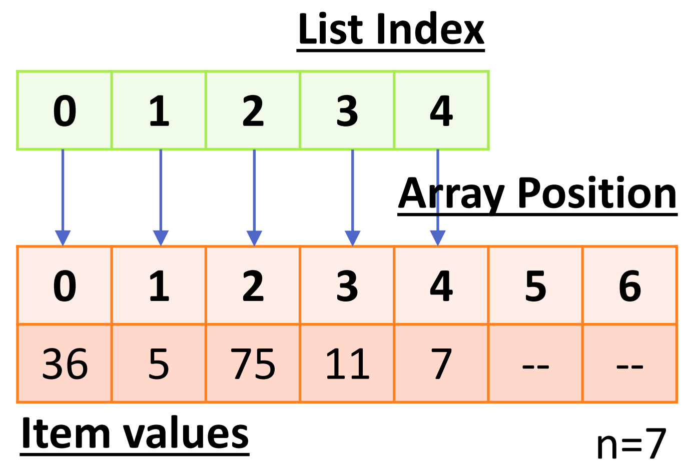
- Operations
  - Create, destroy
  - Add, remove item
  - Lookup by index, item value
  - Find size, check if empty
  - *Precise name of operation may vary with language, but semantics remain same.*

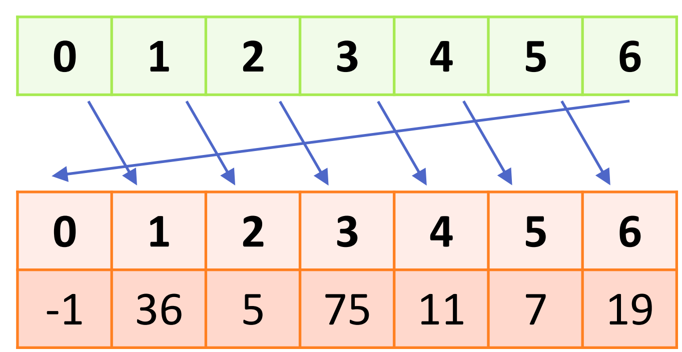
# 1-D Array (implementation of list)

- List implementation using arrays, in a prog. language
  - Array is the **backing** Data Structure
  - Typically limited to **primitive** data types
- Arrays are contiguous memory locations with fixed capacity
  - Contiguous locations mean **locality** matters!
  - **Capacity** is different from **size**. Size is current number of items in list. Capacity denotes max possible size.
- Allow elements of same type to be present at specific positions in the array
  - Position is the **offset** from the start of array memory location, while accounting for **data type size**

# Mapping Function

- Index in a **List** can be mapped to a Position in the **Array**
  - **Mapping function** from index to position
- Say **n** is the capacity of the array
- Simple mapping
  - `position(index) = index`
- Wrap-around mapping (circular buffer)
  - `position(index) = (position(0)+index) % n`
  - `position(0) = front`




n=7, front=1

# List Operations

- item **get**(index)
- item **front**()
- item **back**()

- void **set**(index, item)
- void **append**(item)
- void **remove**(index)

- int **size**()
- int **capacity**()
- boolean **isEmpty**()
- int **indexOf**(item)

# List Operations using Arrays

- `void` **`create`**`(initCapacity)`
  - Create array with initial capacity *(optional hint)*
- `void` **`set`**`(index, item)`
  - Use mapping function to set value at position
  - Sanity checks?
- `item` **`get`**`(index)`
  - Use mapping function to set value at position
  - Sanity checks?

```
class List {        // list with index starting at 1
    int arr[]       // backing array for list
    int capacity    // current capacity of array
    int size        // current occupied size of list

    /**
     * Create an empty list with optional
     * initial capacity provided. Default capacity of 15
     * is used otherwise.
     */
    void create(int _capacity){
      capacity = _capacity > 0 ? _capacity : 15
      arr = new int[capacity]     // create backing array
      size = 0      // initialize size
    }
```

```
  // assuming pos = index-1 mapping fn.
  void set(int index, int item){
    if(index > capacity) { // grow array, double it
      arrNue = int[MAX(index, 2*capacity)]
      // copy all items from old array to new
      // source, target, src start, trgt start, length
      copyAll(arr, arrNue, 0, 0, capacity)
      capacity = MAX(index, 2*capacity) // update var.
      delete(arr) // free up memory
      arr = arrNue
    }
    if(index < 1) {
      cout << "Invalid index:" << index << "Expect >=1"
    } else {
      int pos = index - 1
      arr[pos] = item
      size++
    } // end if
  } // end set()
} // end List
```

**<u>Try yourself!</u>**

Implement **get**(index), **front**() and **back**()

# List Operations using Arrays

- `void` **`append`**`(item)`
  - Insert after current “last” item…use **`size`**
  - Sanity checks?
- `void` **`remove`**`(index)`
  - Remove item at index
  - Sanity checks?
- `int` **`indexOf`**`(item)`
  - Get “first” index of item with given value
  - Sanity checks?

# List Operations using Arrays

- Increasing capacity
- Start with initial capacity given by user, or default
- When capacity is reached
  - Create array with more capacity, e.g. double it
  - Copy values from old to new array
  - Delete old array space
- Can also be used to shrink space
  - Why?
- **Pros & Cons of List using Arrays**

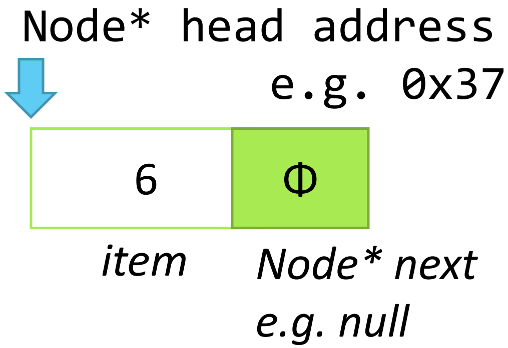
# Linked List Representation

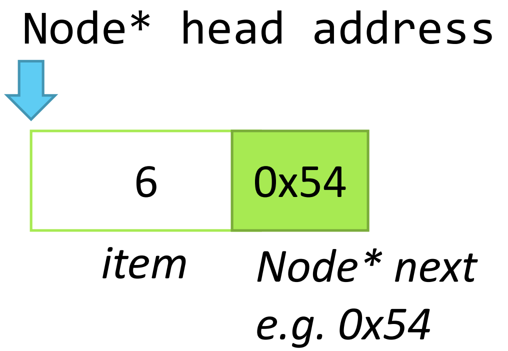
- **Problems with array**
  - Pre-defined capacity, under-usage, cost to move items when full. Fixed-size items (primitives)
- **Solution:** Grow backing data structure dynamically when we add or remove ➲ Only use as much memory as required
- *Linked lists* use ***pointers*** to contiguous chain items
  - **Node** structure contains **`item`** and pointer to **`next`** node in List
  - Add or remove nodes when setting or getting items

**Try yourself!**
Print the values of **pointer locations** for array and linked list items. Is there any pattern?

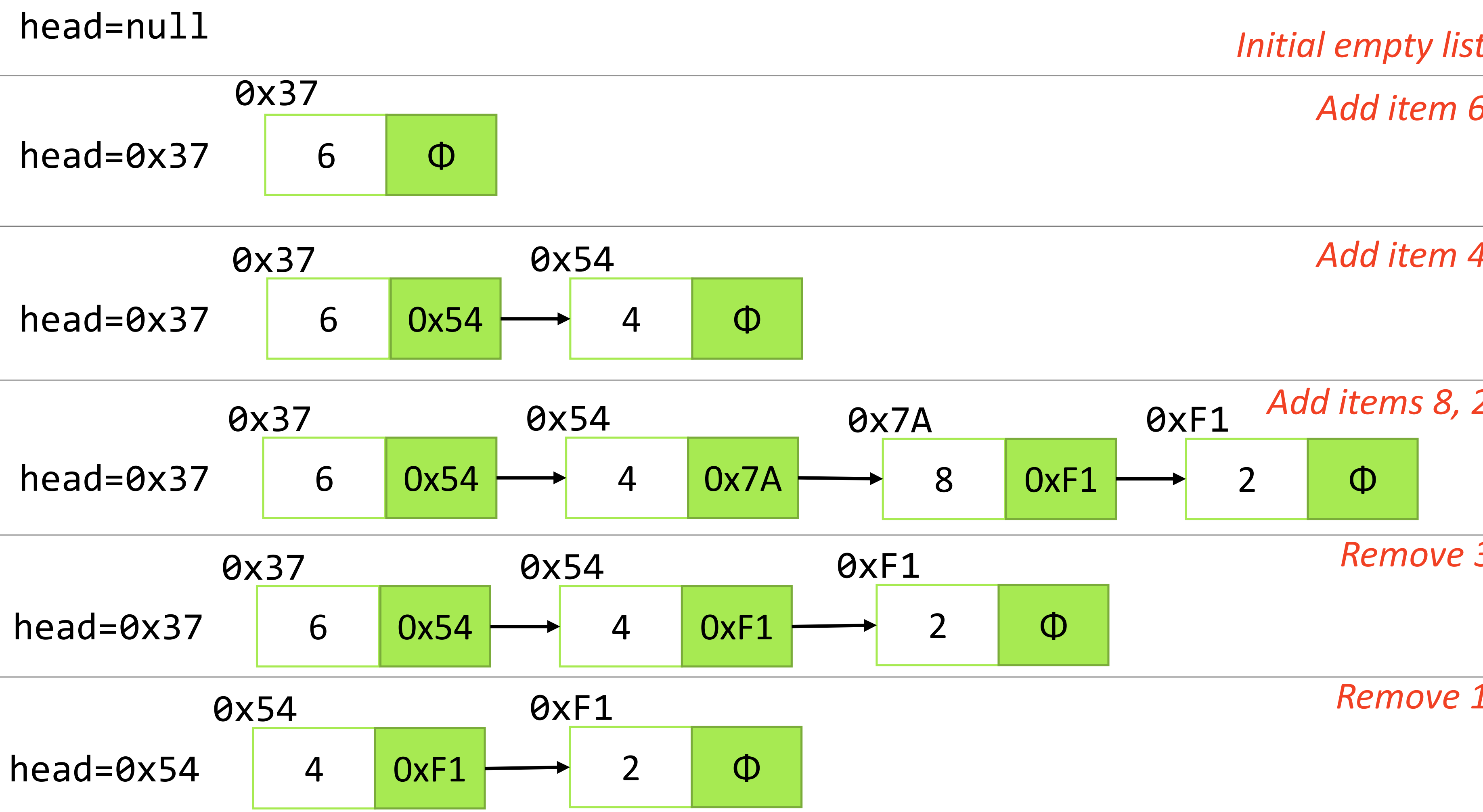
# Node & Chain

```
class Node {
   int item
   Node* next
}
class LinkedList {
   Node* head
   int size
   append() {...}
   get() {...}
   set() {...}
   remove {...}
}
```



# Linked List Operations

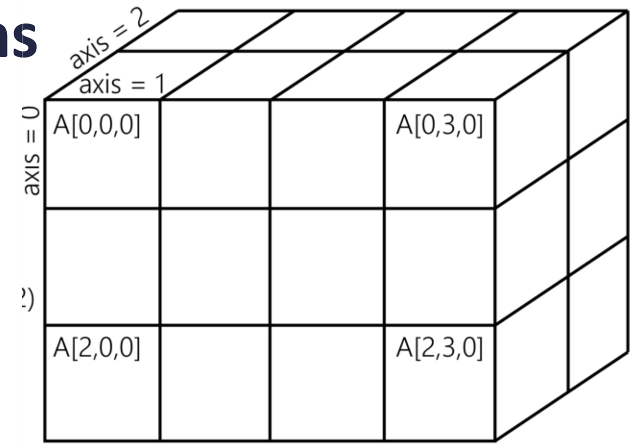


# Matrices & n-D Arrays

# Matrices & n-D Arrays

- Arrays can have more than 1-dimension
  - 2-D Arrays are called **matrices,** higher dimensions called **tensors**
- Arrays have as many **indexes** for access as dimensions
  - A[i], B[i][j], C[i][j][k]
- Dimensions may have **different lengths**

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |




- Mapping from **n-D to 1-D array**
  - Items **n** dimensions “flattened” into **1** dimension
  - Contiguous memory locations in 1-D
  - Native support in programming languages



https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm
https://ipython-books.github.io/13-introducing-the-multidimensional-array-in-numpy-for-fast-array-computations/

# n-D Arrays as 1-D Arrays

- Convert A[i][j] to B[k] … i=row index, j=column index
  - **Row Major Order** of indexing: `k=map(i,j)=i*C+j`
  - **Column Major Order** of indexing: `k=map(i,j)=j*R+I`
- *How does this look in memory location layout?*
- *How can you extend this to higher dimensions (tensors)?*

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 |

(a) Row-major mapping

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 6 | 9 | 12 | 15 |
| 1 | 4 | 7 | 10 | 13 | 16 |
| 2 | 5 | 8 | 11 | 14 | 17 |

(b) Column-major mapping

**Figure 7.2** Mapping a two-dimensional array



*Sahni Textbook, Chapter 7*

# n-D Arrays

- **Array of Arrays** representation
- First find pointer for row array
- Then lookup value at column offset in row array
- *How does this look in memory location layout?*
- *Pros & cons relative to using 1-D array representation?*

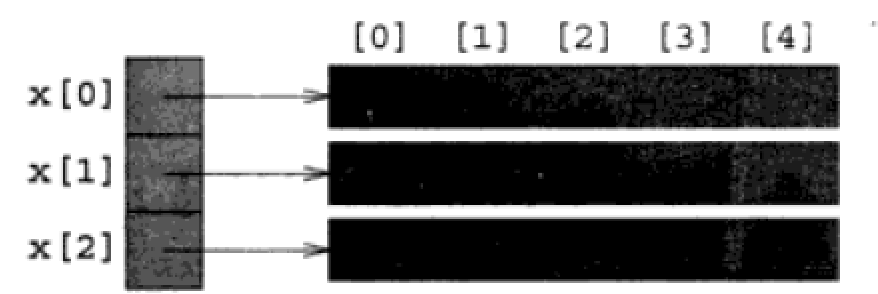


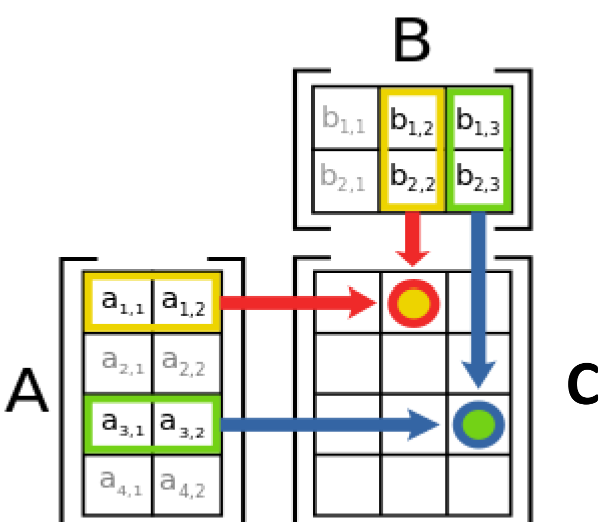

**Figure 7.3** Memory structure for a two-dimensional array



*Sahni Textbook, Chapter 7*

# Matrix Multiplication

```
// Given a[n][n], b[n][n]
// c[n][n] initialized to 0
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
```




https://en.wikipedia.org/wiki/Matrix_multiplication

# Sparse Matrices

- Only a small subset of items are populated in matrix
  - Students and courses taken, faculty and courses taught
  - Adjacency matrix of social network graph
    - vertices are people, edges are “friends”
    - Rows and columns are people, cell has 0/1 value
- Why not use regular 2-D matrix?
  - 1-D representation
  - Array of arrays representation

# Sparse Matrices as 2-D arrays

- Each non-zero item has one entry in list
  - index: <row, column, value>
  - index is the $(i\text{-}1)^{th}$ non-zero item in row-major order
  - Space taken in **3*NNZ** (number of non zero), compared to **n*m** for non-sparse representation

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

(a) A 4 × 8 matrix

| terms | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| row | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 |
| col | 4 | 7 | 2 | 5 | 8 | 4 | 6 | 2 | 3 |
| value | 2 | 1 | 6 | 7 | 3 | 9 | 8 | 4 | 5 |

(b) Its linear list representation

**Figure 7.14** A sparse matrix and its linear list representation



*Sahni Textbook, Chapter 7*

# Sparse Matrix Addition

```
while(p < pMax && q < qMax) {  // C is no. of cols in orig. matrix
   p1 = A[p].r*C + A[p].c      // get index for A in orig. matrix
   q1 = B[q].r*C + B[q].c
   if(p1 < q1)                 // Only A has that index
      C[k] = <A[p].r, A[p].c, A[p].val>                // Copy val
      p++
   else if(p1==q1)             // Both A & B have that index
      C[k] = <A[p].r, A[p].c, A[p].val+B[q].val>       // Add vals
      p++
      q++
   else                        // Only B has that index
      C[k] = <B[q].r, B[q].c, B[q].val>                // Copy vals
      q++
   k++
}
```



*See Sahni, Program 7.17*

# Compressed Sparse Row (CSR)

- Similar to 2-D array, but more **space efficient**
- 3 arrays *(first 2 same as 2-D array representation)*
  - **A[nnz]** stores non-zero values in row-major order
  - **JA[nnz]** stores column index of nnz in A
  - **IA[m+1]** stores cumulative count of non-zero values till (i-1)$^{th}$ row
    - IA[i] = IA[i-1] + number of NNZ in (i-1)$^{th}$ row
    - Always, **IA[0] = 0** and **IA[m+1] = NNZ**
- Space taken = 2*NNZ + (m + 1)
  - How does this compare with full and 2-D sparse representations?

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

```
 A = [ 5 8 3 6 ]
IA = [ 0 0 2 3 4 ]
JA = [ 0 1 2 1 ]
```

**<u>Try yourself!</u>**
Matrix-matrix addition using CSR



https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

# Tasks

- Self study (Sahni Textbook)
  - Chapters 5 & 6 “Linear Lists—Array & Linked Representations”
  - Chapter 7, Arrays and Matrices
- Programming Self Study
  - Try out **list** data structure in **C++ STL**
  - Define your own **abstract list interface** using **templates/generics** in C++. Implement create, set, get, front and back using a **1-d array** representation.
  - Try out **matrix-matrix multiplication** in C++