IS2020 COMP 2540: Data Structures and Algorithms
Lecture 01: Introduction to Data Structures and Algorithms

Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

May 25, 2020

Outline

Why Data Structures & Algorithms?
What is an Algorithm?
Algorithm Analysis and Big-O Notation
Abstract Data Types (ADTs)
Arrays

How are you ?

Write on me

Announcement

- Assignment 1: Due 1 week
- Recoding: I am recording this session for the future reference.

Why Data Structures & Algorithms?

- Web Search in the 90’s

- Search engines ranked pages using keyword frequency
- well-known and worked OK.

Why Data Structures & Algorithms?

- Larry Page & Sergey Brin (PhD students @ Stanford)
  - noticed that links were important too!
  - links convey information about importance
  - But what exactly? and how can you make use of it?
  - This led them to design PageRank

Google

Why Data Structures & Algorithms?

- How does PageRank work?
- Why does it work?
- How do you implement it efficiently?
  - Google indexes hundreds of billions of pages
  - answers and ranks in 0.5 seconds
  - processes 40,000 queries p/s & 3.5 billion queries per day
  - using clever algorithms and data structures!

Google

Google's secret is clever use of algorithms & data structures.

Why Data Structures & Algorithms?

- A Programmer performs following 3 steps:
  - Take some input
  - Process it
  - Give back the output
- Input can be in any form
  - Google Maps: Starting Point, Destination Point
  - Facebook: Username, Password
- After certain processing task, the program gives output in some form.
- To make it more efficient, we need to **optimize** all the three steps.
- The most we can optimize is the 2nd step, which is where we have **Data Structures & Algorithms**.

Why Data Structures & Algorithms?

- Primary Concern: Efficiency(Optimization)
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.
- A solution is said to be efficient if it solves the problem within its resource constraints:
  - Space
  - Time

Why Data Structures & Algorithms?

- Each problem has constraints on available space and time
- Only after a careful analysis of problem characteristics can we know the best data structure for the task.
- **Data Structures:** methods to store information
- **Algorithms:** methods for solving a problem

Impact of Data Structures and Algorithms

Write on me

Impact of Data Structures and Algorithms

- Internet: Web Search, Packet Routing, ...
- Biology: Human Genome Project, Protein Folding, ...
- Computers: Circuit Layout, File System, Compilers, ...
- Computer Graphics: Movies, Video Games, VR, ...
- Security: Cell Phones, E-Commerce, Voting Machines, ...
- Multimedia: MP3, JPG, DivX, HDTV, Face Recognition, ...
- Social networks: Recommendations, News-Feeds, Ads, ...
- Physics: N-body simulation, particle collision simulation, ...

What is an algorithm?

- Well-defined computational procedure.
- A method or a process followed to solve a problem
- An algorithm is a set of precise instructions that transfer the input into the desired output.
- To express an algorithm we could use
  - Natural language
  - Pseudocode
  - Flowcharts
  - Programming language.
- A **computer program** is a concrete representation, for an algorithm in some programming language.

What is a good algorithm?

Any algorithm should have five properties:

- **Finiteness**: The algorithm must always terminate after a finite number of steps.
- **Definiteness**: Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- **Input**: An algorithm has zero or more inputs, taken from a specified set of objects.
- **Output**: An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Effectiveness**: All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

How to decide a good algorithm?

There are two criteria to decide/evaluate an algorithm.

- **Correctness**: the algorithm provide the **desired solution** to the problem in a **finite number of steps**.

- **Efficiency**: how much resources (in term of time and memory) the algorithm consumes to produce the **desired solution** .

Analysis Tools

- Empirical Analysis:

  Implement, Experiment & Calculate the total runtime

  E.g. Current.TimeMills() & nanoTime()

  Drawback?

  A measure of an algorithm's relative to running time, as a function of how many items there are in the input.

  The number of symbols required to reasonably encode the input, which we call n. The n could be:

  - The number of items in a container
  - The length of a string or file
  - The degree of a polynomial
  - The number of digits (or bits) in an integer

Algorithm Analysis

Example: Printing each element of an array

```
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

Here $n = \text{a.length}$

1 initialization of i

n comparisons of i against a.length

n increments of i

n array indexing operations (to compute a[i])

n invocations of System.out.println

so we write $T(n) = 4n + 1$.

Algorithm Analysis

Example: Multiplying two square matrices

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        double sum = 0;
        for (int k = 0; k < n; k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}
```

we need to do is count the number of sum $+=$ a[i][k]*b[k][j] operations — that's really what counts. So we have $T(n) = n^3$. → *We counted operations relative to the width of the matrices.*

We can also count operations relative to the number of items in the two matrices, which is n^2.

Algorithm Analysis: Your Turn!

- If statement:

```
if (a.length > 0) {
   return a[a.length - 1];
} else {
   throw new NoSuchElementException();
}
```

- Addition:

```
x + y
```

- Nested Loop:

```
for (int i = 1; i <= n; i *= 2) {
   for (int j = 0; j < n; j++) {
      count++;
   }
}
```

Algorithm Analysis

- Asymptotic Analysis:

  The asymptotic behavior of a function f(n) refers to the growth of f(n) as n gets large.

  It is the process of determining how processing time increase as the size of the problem increases - time complexity analysis

  Since we are really measuring growth rates, we usually ignore:

  all but the “largest” term, and

  any constant multipliers

Algorithm Analysis

- Best, Worst, and Average
  - **Best Case:** The minimum number of steps taken. (the least time)
  - **Worst Case:** The maximum number of steps taken. (the longest time)
  - **Average Case:** Running the algorithm many times by generating different inputs from some distribution and measure the average time.

Big-O: Asymptotic Upper Bounds

- The execution time is expressed using a function f(n), and the Big-O notation gives the upper bound of f(n)
- The Big-O notation express how the function F(n) behaves as n moves towards ∞.
- When we design an algorithm our objective is minimize the rate of growth.
- In general, it is represented as $f(n) = O(g(n))$, when n is large, the upper-bound of f(n) is g(n). *i.e.* $f(n) \leq C \cdot g(n)$ whenever $n > k$
- Example:

  If input function is $f(n) = n^2 + 2n$, What will be g(n)?

  $g(n) = 3n^2 \rightarrow O(g(n)) = O(3n^2) = O(n^2)$

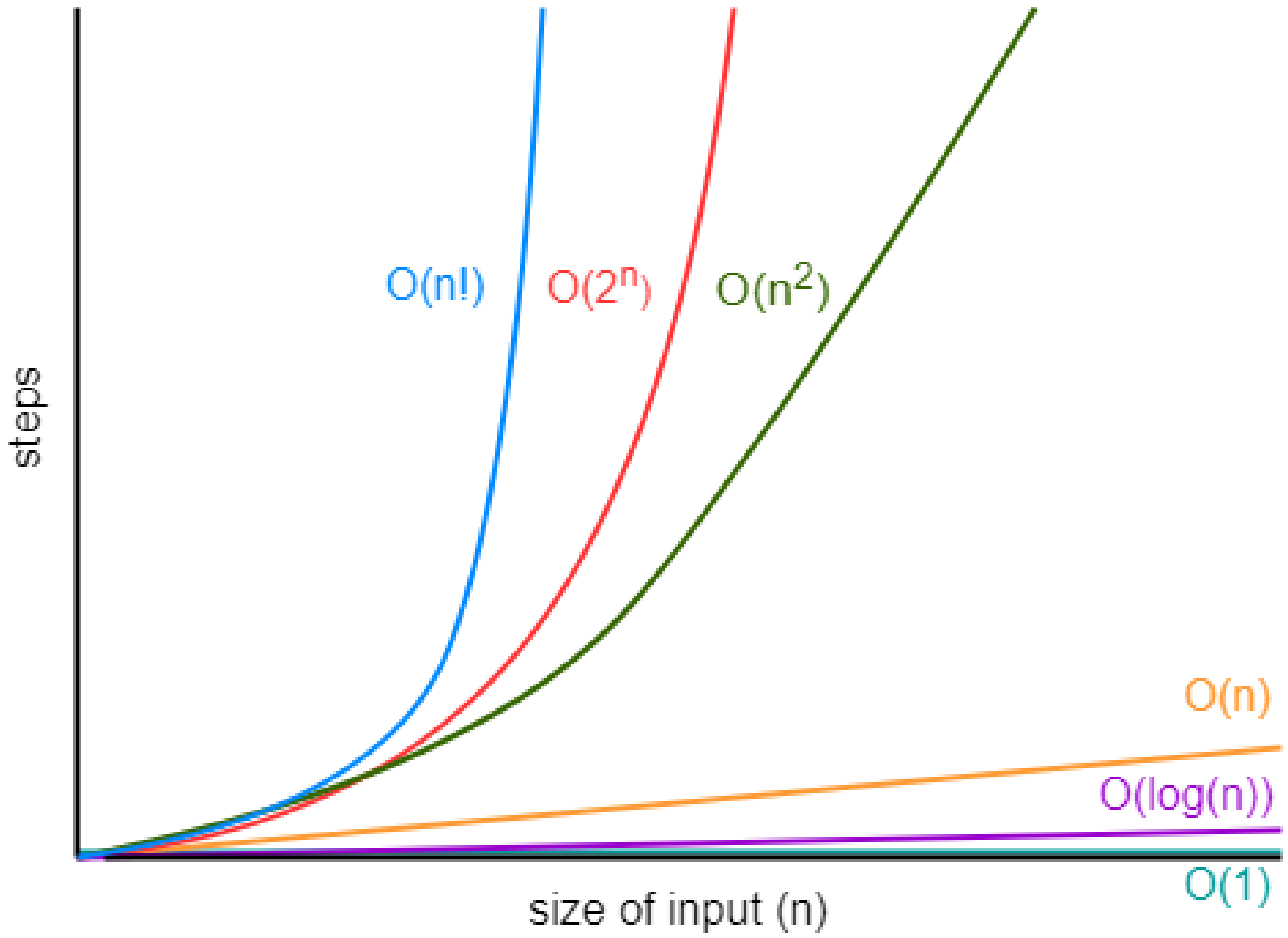
  $\lambda n.0.4n5 + 3n3 + 253$

  $\lambda n.6.22n\log n + n/7$

Big-O Notation

Time Complexity	Name	Examples
$O(1)$	Constant	Inserting an element at the beginning of a linked list.
$O(\log n)$	Logarithmic	Finding an element in a sorted array
$O(n)$	Linear	Finding an element in unsorted array
$O(n \log n)$	Linear Logarithmic	Sorting an array using merge sort or heap sort
$O(n^2)$	Quadratic	Sorting an array using bubble sort or selection sort
$O(c^n)$	Exponential	Calculation of Fibonacci numbers, Towers of Hanoi

Rate of Growth and Big-O Notation



Big-Ω: Asymptotic Lower Bounds

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers.

  We say that f(n) is Ω(g(n))

  – if there are constants C and k such that $f(n) \geq C \cdot g(n)$ when $n > k$

  Lower bounds are useful because they say that an algorithm requires at least so much time.

  Example: $f(x) = 8x^3 + 5x^2 + 7$ is Ω(g(x)) where $g(x) = x^3$

Any Questions?

Data & Data Type

Data	Data Type

Write on me

Data & Data Type

- Data are values or set of values.

28
John Doe
25/11/1990
482.65
What is the weather today?

- Data Type: type of data used in computation

Primitive: Byte, short, int, long, float, char, double, boolean

Class: user defined
People, School, etc

Abstract Data Types (ADTs)

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT only mentions **what operations** are to be performed but **not how** these operations will be implemented.
- Think of ADT as a black box which hides the inner structure and design of the data type.
- ADTs: Array, List, Map, Queue, Stack, Tree etc
- Each ADT operation is defined by its inputs & outputs

Abstract Data Types (ADTs)

- Good Programs Use Abstraction.
- What makes a program good?
  - It works (as specified!).
  - It is easy to understand and modify.
  - It is reasonably efficient.

use **abstract data types**, or **ADTs**

- Benefits of using Abstract Data Types
  - Code is easier to understand
  - Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.
  - ADTs can be reused in future programs.

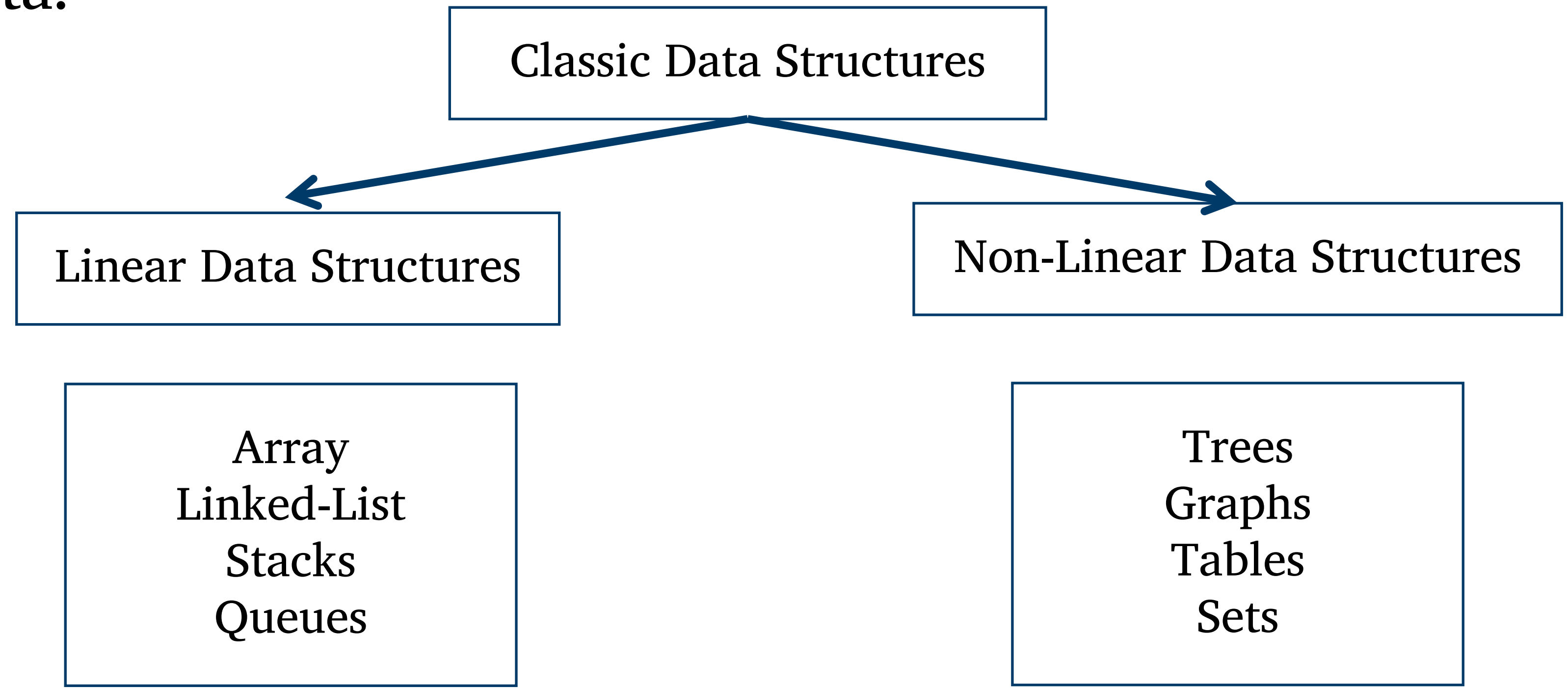
Abstract Data Types (ADTs)

- Advantages of ADTs:
  - Encapsulation: Access by well-defined interface
  - Representation Independence
  - Modularity: Independent development, re-use

Data Structures

- Implementation of an ADT:
  - Each operation associated with the ADT is implemented by one or more methods in the implementation

- Data structure usually refers to an organization for data in main memory
- Data structures are categorized into two types based on how we access the data:



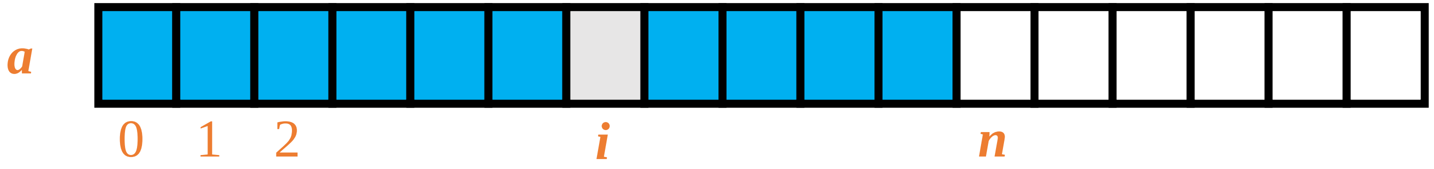
Data Structures Operations

- **Traversing**: access each data element exactly once so that it can be processed.
- **Inserting**: to add a new element to the structure.
- **Deleting**: to remove an existing element from the structure
- **Searching**: to find an element with a given key value or the location of the element in the structure.
- **Sorting**: arranging the elements in the structure in some order.
- **Merging**: to combine two or more structures into one structure.

Arrays

Arrays

- Simple Linear Data Structure.
- An array is a sequenced collection of variables **all of the same type**. Each variable, or cell, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, a , are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an element of that array.



Arrays

35	25	14	16	56

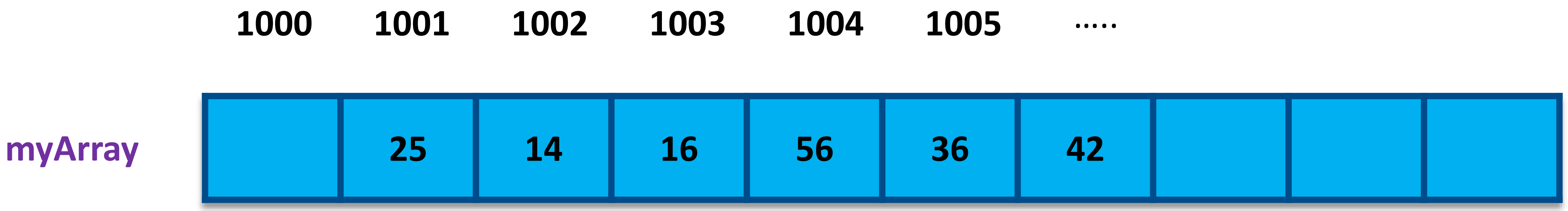
35	25	"A"	"B"	56

Arrays

- Representation of Array in a Memory:
  - The process to determine the address in a memory:

    a) First address – base address. (**Base(myArray)**)

    b) Relative address to base address through index function.



Because the array is allocated in contiguous memory cells, the computer does not need to keep track of the address of every element.

Declaration and initialization:

In Java

1. First method:

$$elementType[\,]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

$$\text{int[] myArray} = \{1,2,3,4,5,6,7,8,9,10\};$$

2. Second method:

new *elementType[length]*

In C/C+ +

int myArray[size];

myArray[3] = 25;

Arrays in C (1/2)

```c
#include <stdio.h>
#define MAX 6

int scanArray(double [], int);
void printArray(double [], int);
double sumArray(double [], int);

int main(void) {
  double list[MAX];
  int size;

  size = scanArray(list, MAX);
  printArray(list, size);
  printf("Sum = %f\n",
         sumArray(list, size));

  return 0;
}
```

```c
// To read values into arr and return
// the number of elements read.
int scanArray(double arr[], int
max_size) {
  int size, i;

  printf("How many elements? ");
  scanf("%d", &size);
  if (size > max_size) {
    printf("Exceeded max; you may only
enter");
    printf(" %d values.\n", max_size);
    size = max_size;
  }
  printf("Enter %d values: ", size);
  for (i=0; i<size; i++) {
    scanf("%lf", &arr[i]);
  }
  return size;
}
```

Arrays in C (2/2)

```
// To print values of arr
void printArray(double arr[], int size) {
 int i;

 for (i=0; i<size; i++)
  printf("%f ", arr[i]);
 printf("\n");
}

// To compute sum of all elements in arr
double sumArray(double arr[], int size) {
 int i;
 double sum = 0.0;

 for (i=0; i<size; i++)
  sum += arr[i];
 return sum;
}
```

Arrays in Java

- In Java, array is an object.
- Every array has a public length attribute (it is not a method!)

TestArray1.java

```
public class TestArray1 {

  public static void main(String[] args) {
    int[] arr; // arr is a reference

    // create a new integer array with 3 elements
    // arr now refers (points) to this new array
    arr = new int[3];

    // using the length attribute
    System.out.println("Length = " + arr.length);

    arr[0] = 100;
    arr[1] = arr[0] - 37;
    arr[2] = arr[1] / 2;
    for (int i=0; i<arr.length; i++)
      System.out.println("arr[" + i + "] = " + arr[i]);
  }
}
```

Declaring an array:
`datatype[] array_name`

Constructing an array:
`array_name = new datatype[size]`

Accessing individual array elements.

```
Length = ?
arr[0] = ?
arr[1] = ?
arr[2] = ?
```

Arrays Operations: Traversing

- Traversing operation means visit every element once.
  - e.g. to print, etc.

A: array, N: array size

1. Set $J = 0$ // J is a counter
2. Repeat Steps 3 and 4 while $J <= N$:
3. Print A[J]
4. Set $J = J + 1$

Arrays Operations: Traversing in Java

- Alternative loop syntax for accessing array elements
- Illustrate toString() method in Arrays class to print an array

TestArray2.java

```
public class TestArray2 {
   public static void main(String[] args) {
      // Construct and initialise array
      double[] arr = { 35.1, 21, 57.7, 18.3 };

      // using the length attribute
      System.out.println("Length = " + arr.length);

      for (int i=0; i<arr.length; i++) {
         System.out.print(arr[i] + " ");
      }
      System.out.println();

      // Alternative way
      for (double element: arr) {
         System.out.print(element + " ");
      }
      System.out.println();
      System.out.println(Arrays.toString(arr));
   }
}
```

```
Length = 4
35.1 21.0 57.7 18.3
35.1 21.0 57.7 18.3
[35.1, 21.0, 57.7, 18.3]
```

Syntax (enhanced for-loop):

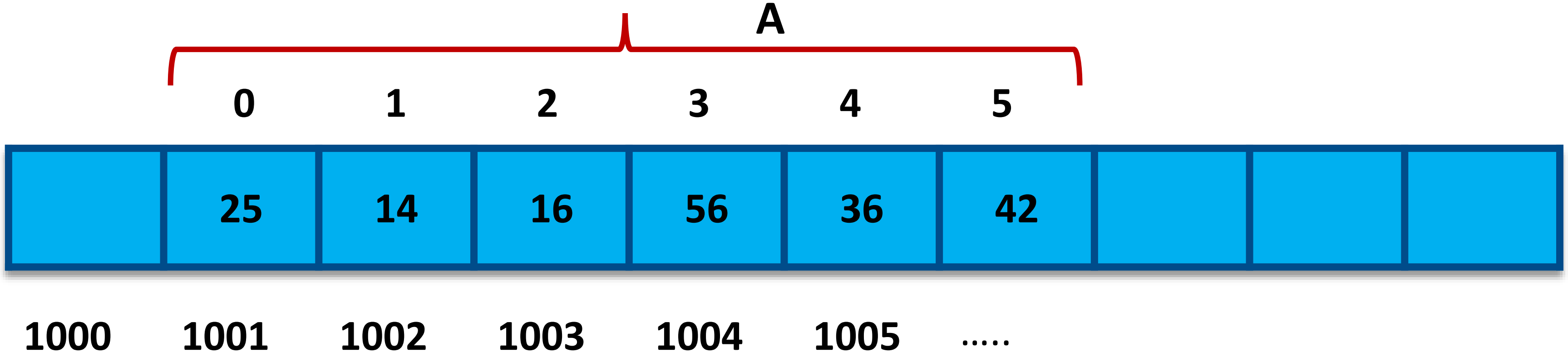
```
for (datatype e: array_name)
```

Go through all elements in the array. “e” automatically refers to the array element sequentially in each iteration

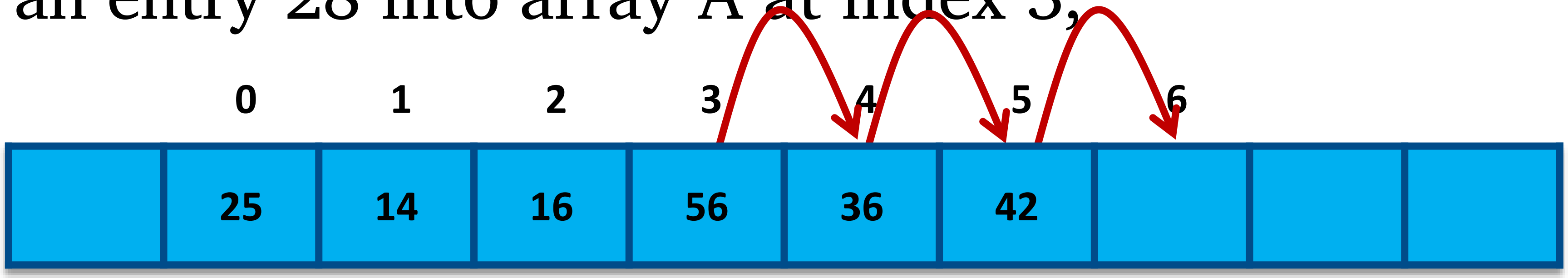
Using toString() method in Arrays class

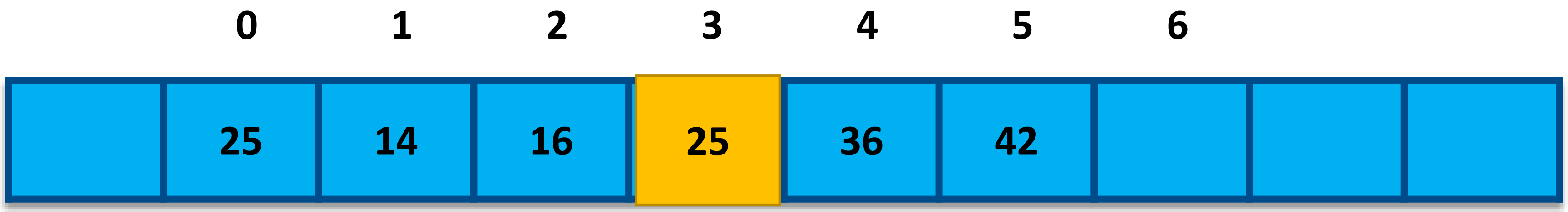
Arrays Operations: Inserting

- Consider the following array.



- Insert an entry 28 into array A at index 3,





Array Operations: Inserting

A: array, N: array size, K: insertion position, V: value to insert

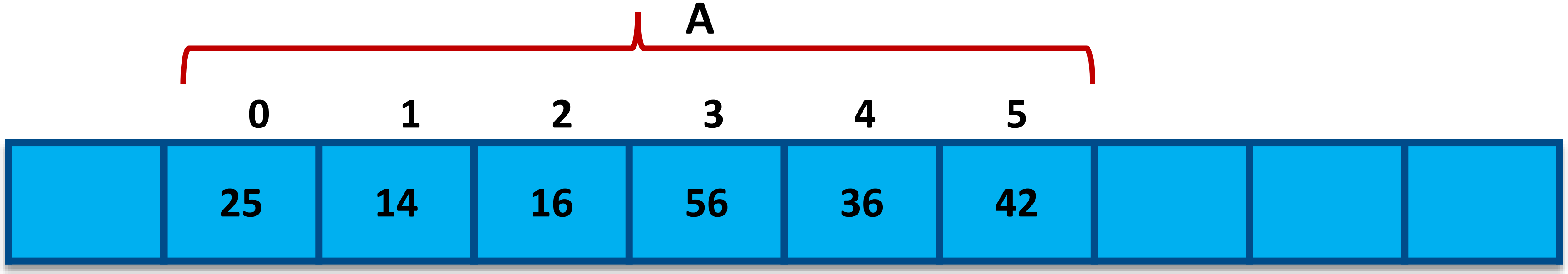
1. Set J = N // J is a counter
2. Repeat Steps 3 and 4 while J > = K:
3. Set A[J+1] = A[J] // move Jth element downward
4. Set J = J –1
5. Set A[K] = V // insert the new element
6. Set N = N+1 // update the size of the array.

Array Operations: Inserting in Java

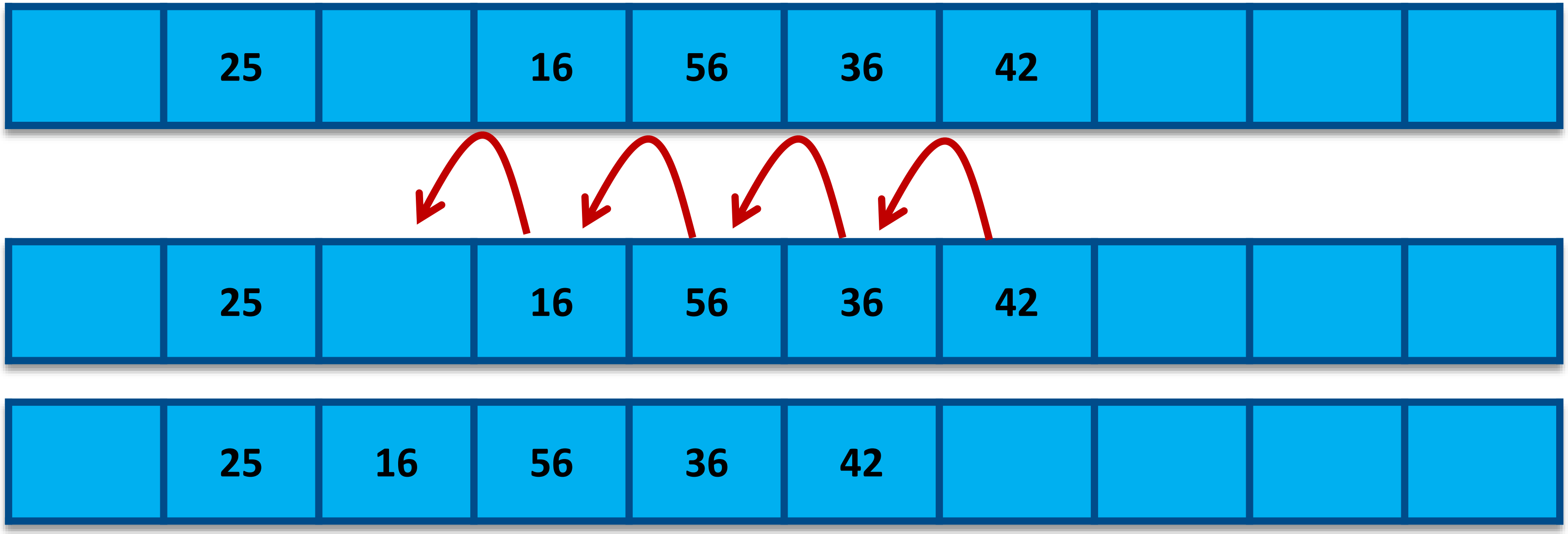
```
public class ArrayInsert {
    public static void main(String arg[]){
      int MAX=10;
      int [] my_array = new int[MAX];
        my_array[0]=10;
        my_array[1]=20;
        my_array[2]=30;
        my_array[3]=40;
        my_array[4]=50;
        my_array[5]=60;

// Insert an element in 3rd position of the array (index->2, value->5)
    int Index_position = 2;
    int newValue    = 5;

    System.out.println("Original Array : "+Arrays.toString(my_array));
    for(int i=my_array.length-1; i > Index_position; i--){
        my_array[i] = my_array[i-1];
    }
    my_array[Index_position] = newValue;
    //print the array after inserting an element
    System.out.println("New Array: "+Arrays.toString(my_array));

}
}
```

Arrays Operations: Deleting at given index

- Consider the following array.



- Delete an entry at index 1,



Arrays Operations: Deleting at given index

A: array, N: array size, K: deletion position

1. Repeat for J = K to N-1:
2. A[J] = A[J+1] // move the J +1st element upward
3. Set J = J+1
4. Set N = N-1 // update the size of the array

Arrays Operations: Deleting at given index in Java

```
public class ArrayDelete {
    public static void main(String arg[]) {
        int MAX = 10;
        int[] my_array = new int[MAX];
        my_array[0] = 10;
        my_array[1] = 20;
        my_array[2] = 30;
        my_array[3] = 40;
        my_array[4] = 50;
        my_array[5] = 60;
        // Delete an element in 3rd position of the array (index->2)
        int Index_position = 2;

        System.out.println("Original Array : "+ Arrays.toString(my_array));
        for(int i=Index_position; i < my_array.length-1; i++){
            my_array[i] = my_array[i+1];
        }
        //print the array after inserting an element
        System.out.println("New Array: "+Arrays.toString(my_array));
    }
}
```

Arrays Operations: Deleting a certain element

Your turn!

Array Operations: Searching

A: array, N: array size, ITEM: search key

1. Set J = 0 // J is a counter
2. F = 0 // flag
3. Repeat Steps 3 and 4 while J < = N:
4. If A[J] equal ITEM then F = 1 break
5. Set J = J –1
6. Print F and J

Array Operations: Searching

```
import java.util.Arrays;
public class ArraysSearching {
    public static void main(String arg[])
    {
        int [] my_array = {10,20,30,40,50,60,70,80};
        // Search an element value= 40
        int searchValue=40;
        boolean flag=false;
        int IndexPosition=0;
        System.out.println("Original Array : "+ Arrays.toString(my_array));
        for(int i=0; i < my_array.length; i++){
            if(my_array[i]==searchValue) {
                flag=true;
                IndexPosition=i;
            }
        }
        if(flag)
            System.out.println("Value 40 is located at index" + IndexPosition);
        else
            System.out.println("NOT FOUND");
    }
}
```

Array Operation: Updating

- Updating an existing element from the array at a given index.

A: array, N: array size, K: updating position

1. Start
2. Set A[K] = ITEM
3. Stop

Arrays

Advantages	Disadvantages

Self Assessment

- What is the run-time complexity of adding an item to an array?
- What is the run-time complexity of deleting an item from an array?
- What is the worst-case and best-case complexity for finding an item in an unsorted array?
- What is the run-time complexity for adding an item to end of an array?