

CSEN 301– Data Structures and Algorithms

Lecture 1: Introduction to Arrays

Prof. Dr. Slim Abdennadher

`slim.abdennadher@guc.edu.eg`

German University Cairo, Department of Media Engineering and Technology

10.09.2017 - 14.9.2017

Course structure

- Lectures
- Exercises and homework
 - Practical Assignments
 - Work in teams, use feedback from tutors
- Labs
 - Supervised lab Assignments
 - Work in teams

WWW-page: Useful info and important announcements

met.guc.edu.eg

Course Regulations

- Weekly graded lab assignments for random groups.
- For attendance, you should commit to your assigned class.
- You should start working on the lab assignment before the lab.

Tentative grading

Overall weighting of your grades:

- 10% for in-lab assignments.
- 20% for quizzes
- 25% for mid-term exam
- 45% for final exam

Survival guide

Tell me and I will forget;
show me and I may remember;
involve me and I will understand.

Keep up with the course material

- Attend lectures, tutorials, and labs
- Participate in the discussions (be active)
- Solve the assignments and understand the model answers provided

WWW-page

Visit course home page regularly for announcements and supplemental material

met.guc.edu.eg

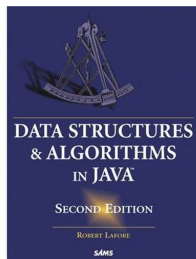
Aims of this course

- To study the concept of **algorithms** and the concept of **algorithm efficiency**.
- To study the concept of **abstract data types** (ADTs) and the abstract data types most commonly used in software development (*stacks*, *queues*, *lists*, *sets*, ...).
- To study the **basic data types** most commonly used to represent these abstract data types (*arrays*, *linked lists*, *binary search trees*, ...), together with **algorithms** operating on these data structures.
- **Java implementation** of common algorithms, ADTs, and data structures.

Learning resources

- **Textbook** for this class:

Robert Lafore. *Data Structures & Algorithms in Java*, Sams Publishing, 2003



- **Resources** at course website at met.guc.edu.eg

The need for efficient data structures and algorithms

- **Representing information** is fundamental to computer information processing
- Computer programs should organize their data in a way that supports **efficient processing**.
- For this reason the **study of data structures and algorithms** that manipulate them is **essential** to computer science and software engineering.
- Knowing appropriate ADTs and algorithms for particular problems would **save** your **programming time** as well as **ensure** the **efficiency of your software**.

Algorithms vs. programs

● Algorithm

- Can be performed by **humans** or by **machines**
- Can be expressed in **any** suitable **language**
- Can be on **any level of abstraction**

● Program

- Must be performed by a **machine**
 - Must be expressed in a **programming language**
 - Must be **detailed** and **specific** to **avoid any ambiguities**
-
- To employ an algorithm on a **machine**, it has to be **transcribed into a programming language** (*i. e.*, **coded!**).
 - There may be **many ways** of coding the algorithm and there is **a wide choice** of **programming languages**. But all the resulting programs are **implementations** of the same underlying algorithm.
 - Here we express our implementations in **Java**

Data structures

- A **data structure** is a systematic way of organizing a collection of data.
- A **static** data structure is one whose capacity is fixed at creation (e. g., *array*).
- A **dynamic** data structure is one whose capacity is variable, so it can expand or contract at any time (e. g., *linked list*, *binary tree*).
- For each data structure, we need **algorithms** for insertion, deletion, searching, *etc.*

Arrays – General properties

- An **array** is an indexed sequence of components

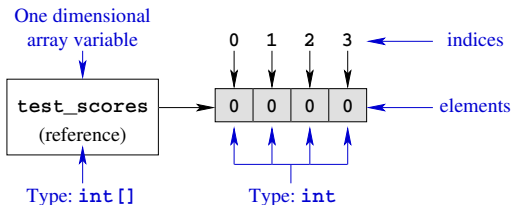
data	
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

- The **length** of an array is **fixed** when the array is constructed.
- Each array component has a **fixed** and **unique index**. The indices range from a **lower bound** to an **upper bound**.

Java primitive arrays

- Code to **create** a one-dimensional array that stores data items based on a primitive type:

```
int [] test_scores = new int [4];
```



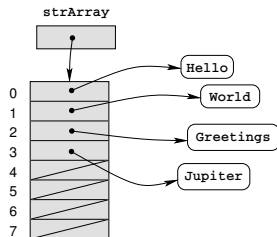
- Code to **initialize** an array of integers:

```
int [] test_scores = new int [] { 70, 80, 20, 30 };
```

Arrays of objects

- We have only considered arrays of elements of the various **primitive types**.
- But we can also have **arrays of objects** of any class, e. g.

```
String[] strArray = new String[8];  
strArray[0] = "Hello" ;  
strArray[1] = "World" ;  
strArray[2] = "Greetings" ;  
strArray[3] = "Jupiter" ;
```



- More generally we can have object arrays:

```
Object[] heap = new Object[20];
```

Example

Example: Writing a **String** to the monitor

```
class StringArray {
    public static void main ( String[] args )
    {
        string[] strArray = new String[8] ;
        strArray[0] = "Hello" ;
        strArray[1] = "World" ;
        strArray[2] = "Greetings" ;
        strArray[3] = "Jupiter" ;
        strArray[ strArray.length-1 ] = "the_end" ;
        for (int j=0; j < strArray.length; j++ )
            if ( strArray[j] != null )
                System.out.println( "Slot_" + j + ":_ " + strArray[j] );
            else
                System.out.println( "Slot_" + j + ":_ " + "empty" );
    } }
```

Unordered array

- **Insertion:** Use the normal array syntax, *e. g.*

```
arr[0] = 5;
```

- **Searching:**
 - Step through the array, comparing the item with each element.
 - If the loop variable reaches the last occupied cell with no match being found, the item is not in the array.
- **Deletion:**
 - Search for the specified item
 - Move all the items with higher index values down one element to fill in the “hole” left by the deleted element.

Time complexity for unordered arrays

- Insertion:

$O(1)$

- Searching:

$O(n)$

- Deletion:

$O(n)$

Sorted arrays

- An array is **sorted** if its components are in ascending order (*i. e.*, each component is less than or equal to the component on its right).
- The meaning of the comparison “*x is less than y*” must be defined for each data type:
 - Meaning of **less for numbers**:
x is **numerically less** than *y* (*i. e.*, $x < y$).
 - Conventional meaning of **less for strings**:
x precedes *y* **lexicographically** (*e. g.*, “**p**ool” is less than “**p**oor”, which is less than “**p**op”).

Insertion and deletion

- **Insertion:**

- Suppose we want to insert the value **8** into this sorted array (while keeping the array sorted).

1	3	4	7	9	12					
---	---	---	---	---	----	--	--	--	--	--

- We can do this by shifting all the elements after the mark right by one location.

1	3	4	7	9	→	12				
---	---	---	---	---	---	----	--	--	--	--

1	3	4	7	→	9	12				
---	---	---	---	---	---	----	--	--	--	--

1	3	4	7	8	9	12				
---	---	---	---	---	---	----	--	--	--	--

- **Deletion:** Deleting an element is similar to the unordered arrays.

Binary search – The idea

How do we look up words in a list that is already sorted?

- Dictionary
- Phone book

Method

- 1 Open up the book roughly in the middle
- 2 Check in which half the word is.
- 3 Split that half again in two.
- 4 Continue until we find the word.

Binary search – An example

position:	0	1	2	3	4	5	6
content:	Amal	Amira	Engy	Mohammed	Noura	Rehab	Slim

We are searching **Engy**

- 1 The midpoint between 0 and 6 is **3**
- 2 We compare **Mohammed** with **Engy**
- 3 Engy **precedes** Mohammed
- 4 We continue the search on the **front half** of the list
- 1 The midpoint between 0 and 2 is **1**
- 2 We compare **Amira** with **Engy**
- 3 Engy **follows** Amira
- 4 We continue the search on the **back half** of the list

Binary search – An example

position:	0	1	2	3	4	5	6
content:	Amal	Amira	Engy	Mohammed	Noura	Rehab	Slim

- 1 The midpoint between 2 and 2 is 2
- 2 We compare Engy with Engy
- 3 Engy equals Engy
- 4 We have found the entry.

Example

```
public int find(long searchKey) {
    int lowerBound = 0;
    int upperBound = nElems - 1;
    int curIn;
    while(true)
    {
        curIn = (lowerBound + upperBound ) / 2;
        if(a[curIn] == searchKey)
            return curIn;                // found it
        else if(lowerBound > upperBound)
            return nElems;                // can't find it
        else                                // divide range
        {
            if(a[curIn] < searchKey)
                lowerBound = curIn + 1;    // it's in upper half
            else
                upperBound = curIn - 1;    // it's in lower half
        }
        // end else divide range
    }
    // end while
}
// end find()
```

Time complexity for sorted arrays

- Insertion:

$O(n)$

- Searching:

$O(\log n)$

- Deletion:

$O(n)$

Conclusions

- Arrays have the following **advantages**:
 - Accessing an element by its index is very fast.
- Arrays have the following **disadvantages**:
 - All elements must be of the same type.
 - The array size is fixed and can never be changed.
 - Insertion into arrays and deletion from arrays is very slow.