Data Structures and Complexity

Memory, Data Structures and Complexity Notations

O(n)

The Big O

Size of the Input

SoftUni Team

Technical Trainers

Software University

Software University

https://softuni.bg

SoftUni

Table of Contents

Memory Storage

Memory Storage and Hierarchy

What Do We Call Memory?

- Computer **memory** is any physical device capable of storing information temporarily, like **RAM**, or permanently, like **ROM**. Memory devices utilize **integrated circuits** and are used by **operating systems**, **software**, and **hardware**.
- The term "memory", meaning "primary storage" or "**main memory**", is often associated with addressable **semiconductor memory**.

What Do We Call Memory?

- In computer science, memory usually is:
  - a continuous, numbered – aka addressed – sequence of bytes
  - storage for variables and functions created in programs
  - random-access – equally fast accessing **5**th and **500**th byte
  - addresses numbered in hexadecimal, prefixed with `0x`.

Address	`0x0`	`0x1`	`0x2`	`...`	`0x6afe4c`	`...`
Byte$_{(binary)}$	00001101	00101010	01000101	...	00000011	...

Memory Usage by Variables

- A primitive data type takes up a sequence of bytes
  - **byte** is **1** byte, **1** address:

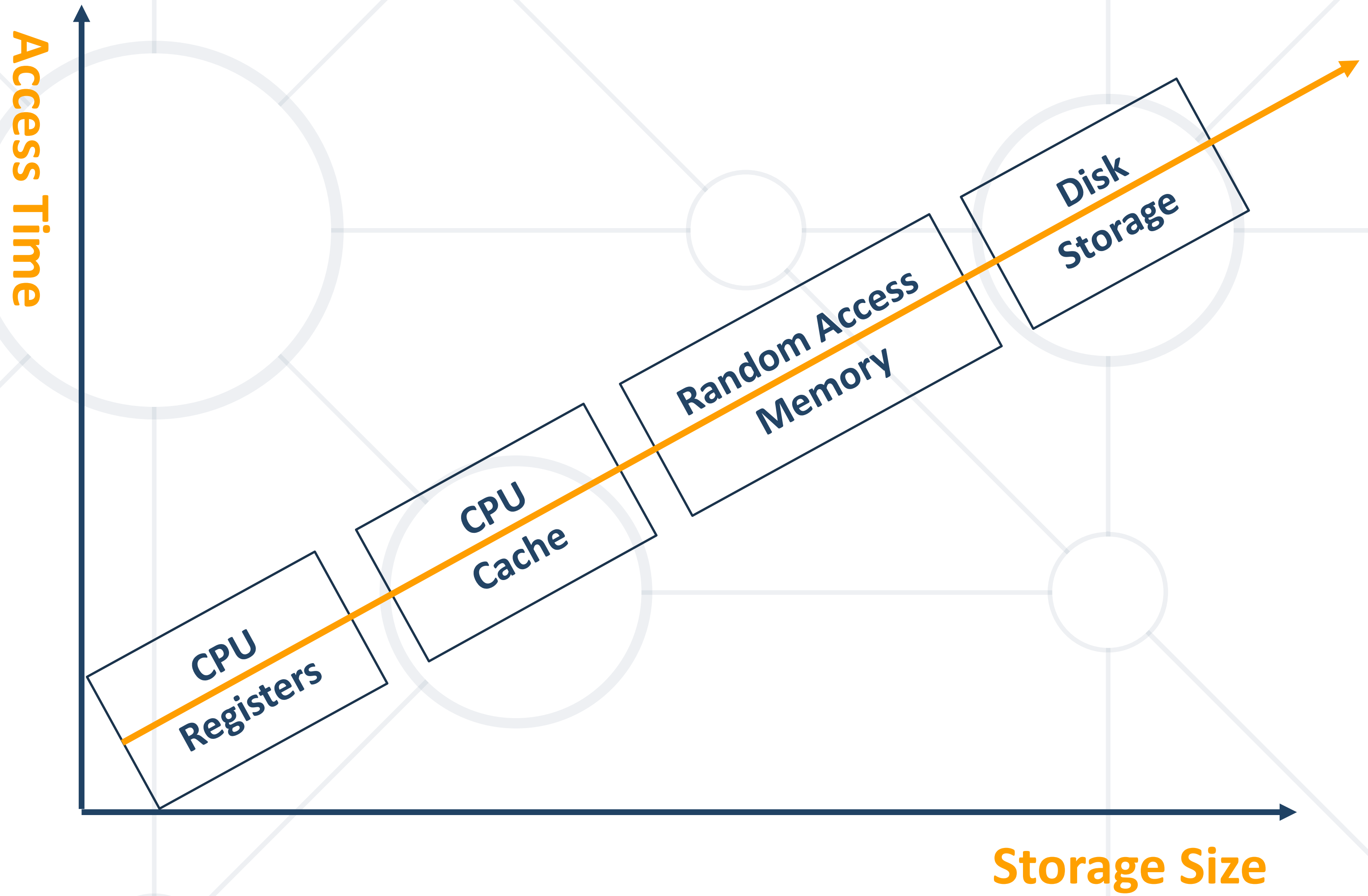
byte number = 42; // let's assume number is at address 0x6afe4c							
Address	...	0x6afe4b	0x6afe4c	...	...	...	...
Byte$_{(binary)}$	...	...	00101010	...	...	...	...

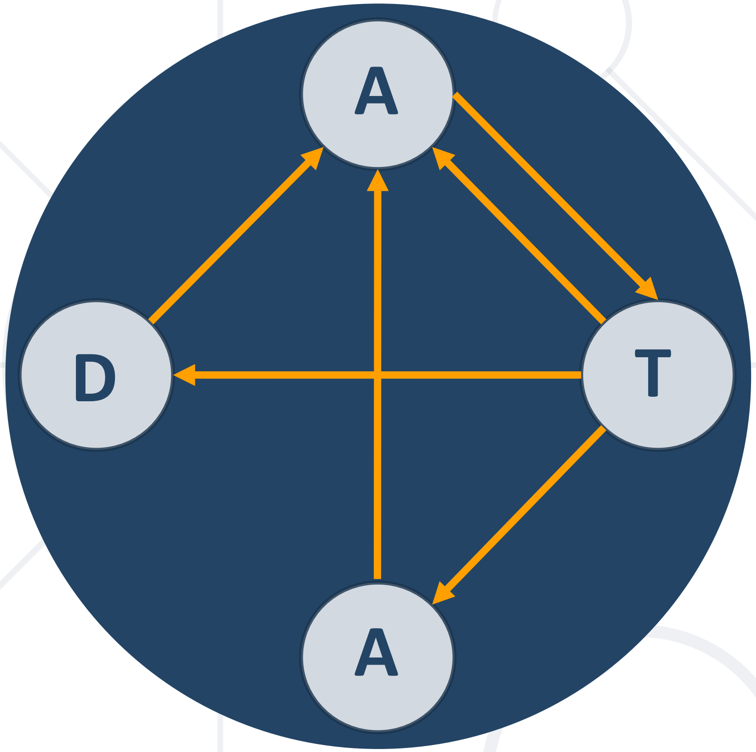
  - Other types & arrays use consecutive bytes, e.g. **4**-byte **int**

int year = 2020; // let's assume year is at address 0x6afe4c							
Address	...	0x6afe4b	0x6afe4c	0x6afe4d	0x6afe4e	0x6afe4f	...
Byte$_{(binary)}$	...	...	11100100	00000111	00000000	00000000	...

Memory Hierarchy

- Each memory level is **faster** and **smaller** than the **next memory level**. At the end we can say we have **nearly infinite memory** storage that **is also infinitely slow**.





Data Structures

Overview

What is Data?

- "**Data**" from Latin – datum, which originally meant "something given". Dates back to the 1600s.
- Data is **raw, unorganized** facts that need to be processed. Data can be something simple and seemingly **random** and **useless** until it is **organized.**
- Example:
  The history of temperature readings all over the world for the past 100 years is data.

What is Information?

- "**Information**" has Old French and Middle English origins. It has always referred to "the act of informing", usually in regard to education, instruction, or other knowledge communication.
- When data is **processed, organized, structured or presented** in a **given context** so as to **make it useful**, it is called **information.**
- Example:
  The history of temperature readings all over the world for the past 100, when organized and analyzed we find that global temperature is rising. – That is information.

Data in Computing

- Set of **symbols** gathered and translated for **some purpose.**
- Simplified – bits of information stored in memory. If those bits remain **unused,** they don't do anything.
- Example:

Binary Data	Translation
100 0001	65
100 0001	A

Data in Computing

Software University

- It is easy to notice, that the way we **read** the data **retrieves the information** of the bits in different ways. However those bits have only **0** or **1** as values.
- Example:

Type	Binary Data	Translation
Integer	0000 0100 0001	65
Character	0000 0100 0001	'A'
Double	0000 0100 0001	65.0
Instruction Code	0000 0100 0001	Store 65
Color	0000 0100 0001	

Data Structures

- Data structure – an **object** which takes responsibility for data **organization**, **storage**, **management** in **effective** manner.
- Storing items **requires memory consumption**:

Data Structure	Size
int	= 4 bytes
float	= 4 bytes
long	= 8 bytes
int[]	≈ (Array length) * 4 bytes
List<Double>	≈ (List size) * 8 bytes
Map<Integer, int[]>	≈ (Map size) * Entry bytes

Abstract Data Structures (ADS)

- An **Abstract Data Structure (ADS)** – the way the real objects will be modulated as **mathematical** objects, alongside the **set of operations** to be executed upon them, **without** the implementation itself.

```
public interface List<E> {
    boolean add(E e);
    int size();
    boolean remove(Object o);
    boolean isEmplty();
}
```

Data Structures Implementation

- An **implementation** – definitive way of ADS to be presented inside the computer memory, alongside the implementation of the operations.

```
public class ArrayList<E> implements List<E> {
    public boolean add(E e) {
        this.elements[this.index++] = e;
        this.size++;
        return true;
    }
}
```

$O(n)$

Algorithmic Complexity

Asymptotic Notation

Algorithm Analysis

- Why should we analyze algorithms?
  - Predict the **resources** the algorithm will need
    - Computational time (**CPU** consumption)
    - Memory space (**RAM** consumption)
    - Communication **bandwidth** consumption
    - **Hard disk** operations

Algorithm Analysis

- There are three main properties we want to analyze:
  - **Simplicity** – this is really a matter of intuition and of course it is subjective quality
  - **Accuracy** – this seems easy to determine, however it may be very difficult to determine is the algorithm correct?
  - **Performance** – the consumption of CPU, Memory and other resources (we really care the most for the first two)

Algorithm Analysis (3)

- The expected **running time** of an algorithm is:
  - The total number of **primitive operations** executed (machine independent steps)
  - Also known as **algorithm complexity**
  - Compare algorithms **ignoring details** such as **language** or **hardware**

Problem: Get Number of Steps

- Calculate maximum steps to find the result

```
long getOperationsCount(int n) {
  long counter = 0;
  for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      counter++;
  return counter;
}
```

Solution:
T(n) = 3(n ^ 2) + 3n + 3

- The input(**n**) of the function is the main source of steps growth

Simplifying Step Count

- Some parts of the equation **grow much faster** than others
  - $T(n) = \mathbf{3(n^2)} + 3n + 3$
  - We can **ignore** some part of this equation
  - Higher terms **dominate** lower terms – $n > 2$, $n^2 > n$, $n^3 > n^2$
  - Multiplicative constants can be **omitted** – $12n \rightarrow n$, $2n^2 \rightarrow n^2$
- The previous solution becomes $\approx n^2$

Time Complexity

- Worst-case
  - An **upper** bound on the running time
- Average-case
  - **Average** running time
- Best-case
  - The **lower** bound on the running time (the optimal case)

Time Complexity

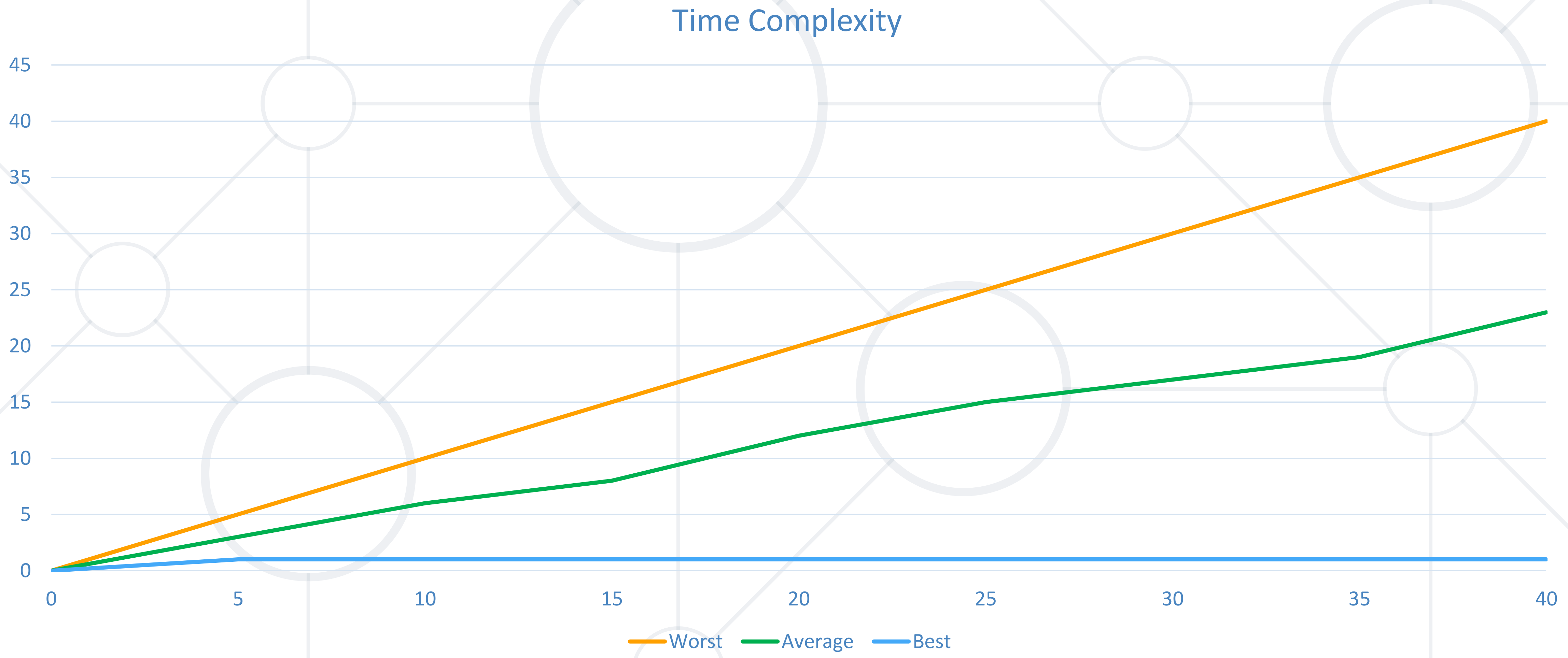
- Define the time complexity of the following code:

```
boolean contains(int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
      if (numbers[i] == number) {
         return true;
      }
  }
  return false;
}
```

- It is not as simple as the previous, **when** does the code return?

Time Complexity

- Therefore, we need to measure **all** the possibilities:



Time Complexity

Software University

- From the previous chart we can deduce:
  - For smaller size of the input (**n**) we **don't care much for the runtime**. So we measure the time as **n** approaches **infinity**
  - If an algorithm **has to scale**, it **should compute** the result within a **finite and practical time**
  - We're concerned about the **order of an algorithm's complexity**, not the actual time in terms of **milliseconds**

Asymptotic notations

- **Asymptotic notations** are descriptions that allow us to examine an algorithm's running time by expressing its **performance** as the input size, **n**, of an algorithm or a function **f increases**. There are **three** common asymptotic notations:
  - Big **O** – **O(f(n))**
  - Big **Theta** – **Θ(f(n))**
  - Big **Omega** – **Ω(f(n))**

Algorithms Complexity

- **Algorithmic complexity** – rough estimation of the number of steps performed by given computation, depending on the size of the input
- Measured with an asymptotic notation
  - `O(f(n))` – upper bound (worst case)
  - `Θ(f(n))` – average case
  - `Ω(f(n))` – lower bound (best case)
    - Where `f(n)` is a function of the size of the input data

Algorithmic Complexity

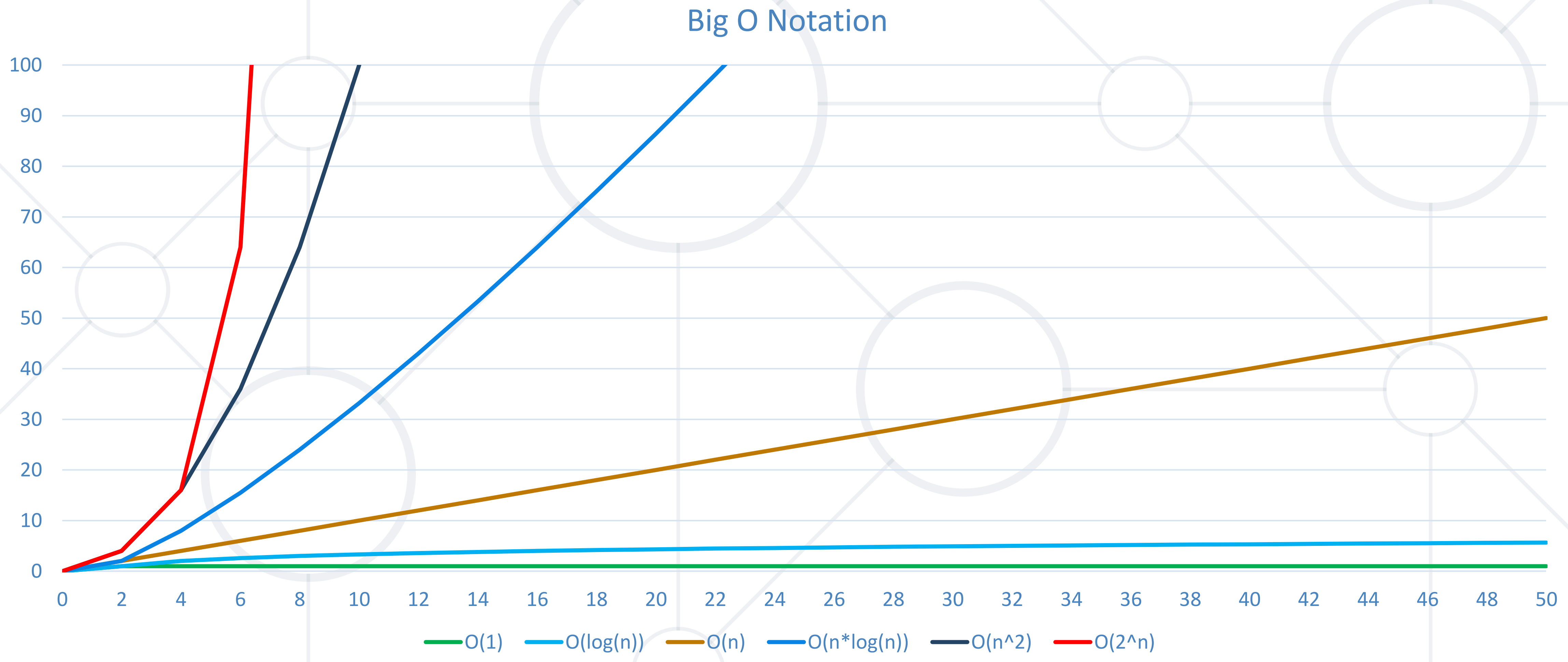
- In this course we will analyze only the Big O – **O(f(n))**

```
boolean contains(int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
      if (numbers[i] == number) {
         return true;
      }
  }
  return false;
}
```

- So the code above will have **O(n)** or simply **linear** complexity

Asymptotic Functions

- Below are some examples of **common algorithmic** grow:



Typical Complexities

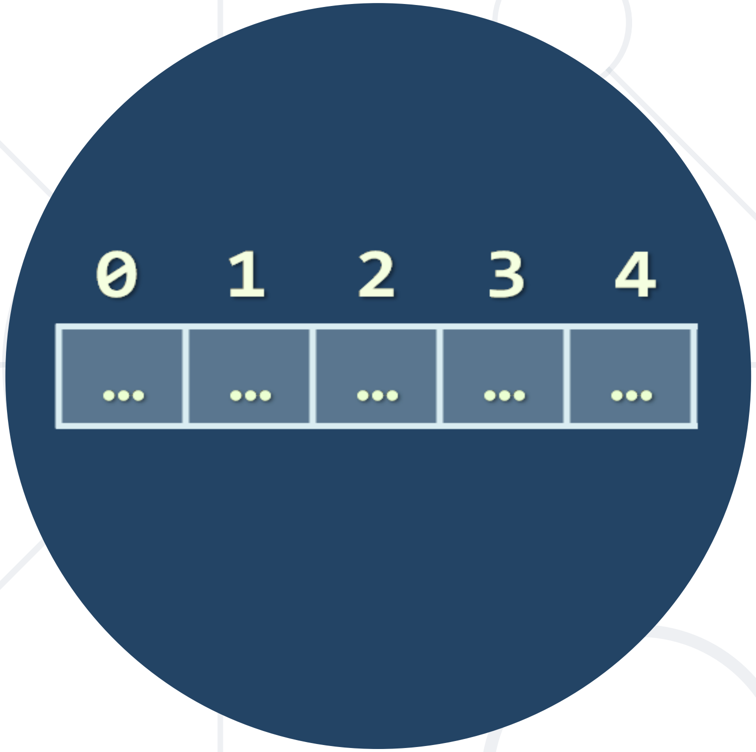
Complexity	Notation	Description
constant	O(1)	n = 1 000 → 1-2 operations
logarithmic	O(log n)	n = 1 000 → 10 operations
linear	O(n)	n = 1 000 → 1 000 operations
linearithmic	O(n*log n)	n = 1 000 → 10 000 operations
quadratic	O(n2)	n = 1 000 → 1 000 000 operations
cubic	O(n3)	n = 1 000 → 1 000 000 000 operations
exponential	O(n^n)	n = 10 → 10 000 000 000 operations

Time Complexity and Program Speed

Complexity	10	20	50	100	1 000	10 000	100 000
O(1)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(log n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n*log n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n^2)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	3-4 min
O(n^3)	< 1 s	< 1 s	< 1 s	< 1 s	20 s	5 hours	231 days
O(2^n)	< 1 s	< 1 s	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 s	hangs	hangs	hangs	hangs	hangs	hangs
O(n^n)	3-4 min	hangs	hangs	hangs	hangs	hangs	hangs

Memory Requirement

- **Memory consumption** should also be considered, for example:
  - Storing elements in a matrix of size N by N
    - Filling the matrix – Running time $O(n^2)$
    - Get element by index – Running time $O(1)$
    - Memory requirement $O(n^2)$
- However in this course we **won't be optimizing** memory consumption we will only point it out



Array Data Structures

Built-in and Lightweight

Array Data Structure

- Ordered
- Very **lightweight**
- Has a **fixed size**
- Usually **built into the language**
- Many collections are implemented by using arrays, e.g.
  - **ArrayList<E>** in Java
  - **ArrayDeque<E>** in Java

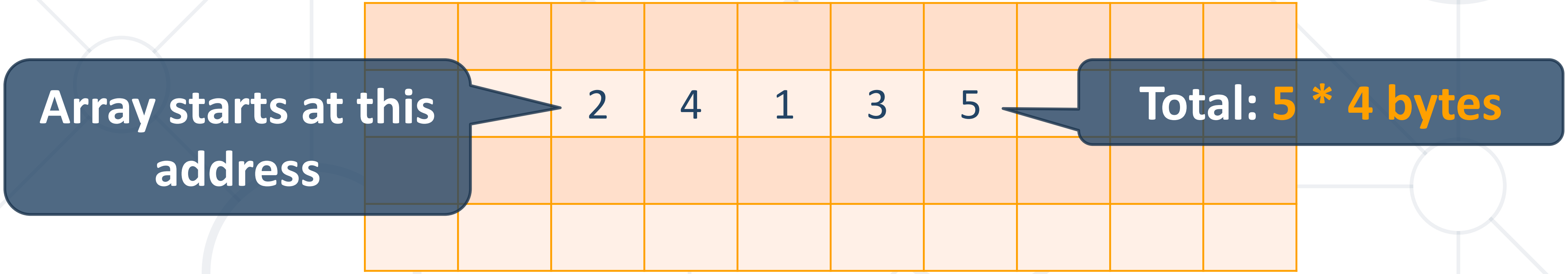
Why Arrays Are Fast?

- Arrays use a **single block of memory**

```
int[] array = { 2, 4, 1, 3, 5 };
```

int size is **4 bytes**

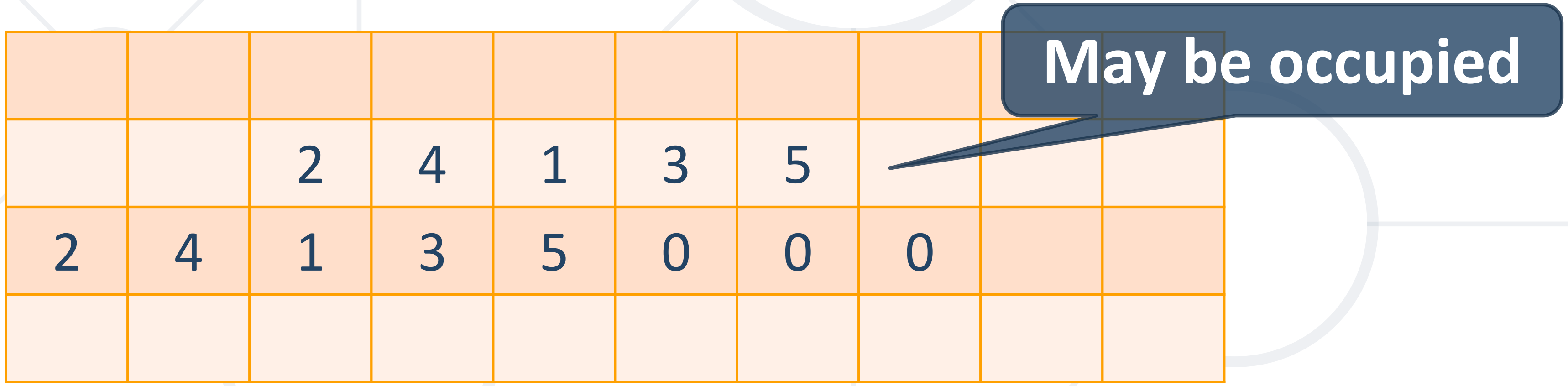
- Uses total of **array pointer + (N * element/pointer size)**



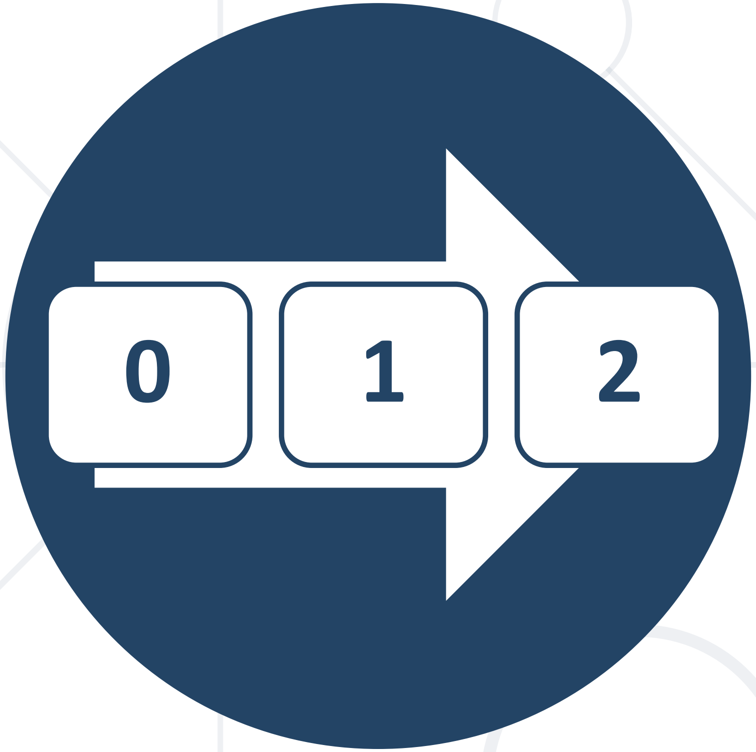
- **Array Address** + **(Element Index** * **Size)** = **Element Address**
- Array Element Lookup – **O(1)**

Arrays – Changing Array Size

- Arrays have a **fixed size**
- Memory after the array **may be occupied**
- If we want to resize the array we have to **make a copy**



- Array Copy – **O(n)**



Data Structure Implementation

Elements Representation Approaches

How Do We Store the Elements?

- **Choose** the way to **store** the elements:
  - By **using an array**:
    - Stores the elements as a **sequence** inside the computer memory
  - By **using a Node<E>** class:
    - Contains the **element** inside the Node. **Must** have **pointer to the next Node.** Can have **more** fields if necessary

Using an Array

- Store the elements

```
public class ArrayList {
    private int[] elements;
}
```

- We can access indices with **O(1)** – constant complexity
- The **rest** operations on **unsorted** arrays are **linear**
- Array initial **size**?
- What happens when we **exceed** the **initial** size?

Using an Array (2)

- Implement **grow()** method when you **need more space**

```java
public class ArrayList {
    private void grow() {
        // Create new array with larger size
       // Copy the elements from the old to the new array
        // Do additional operations if needed
    }
}
```

- What is the complexity? – **O(n)**

Using Node Class

- We can use nested class – **Node**

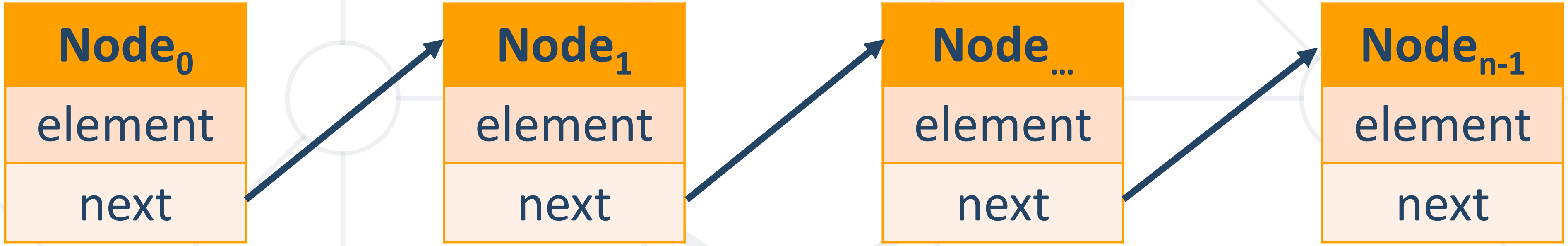
```
public class ArrayList {
    private static class Node {
        private int element;
        private Node next;
        //You can add any fields needed
    }
    private Node head;
}
```

Keep at **least one** reference to connect the nodes

- How to **connect** the **sequence of elements**?

Using Node Class (2)

- Keep **chaining** elements **when adding**



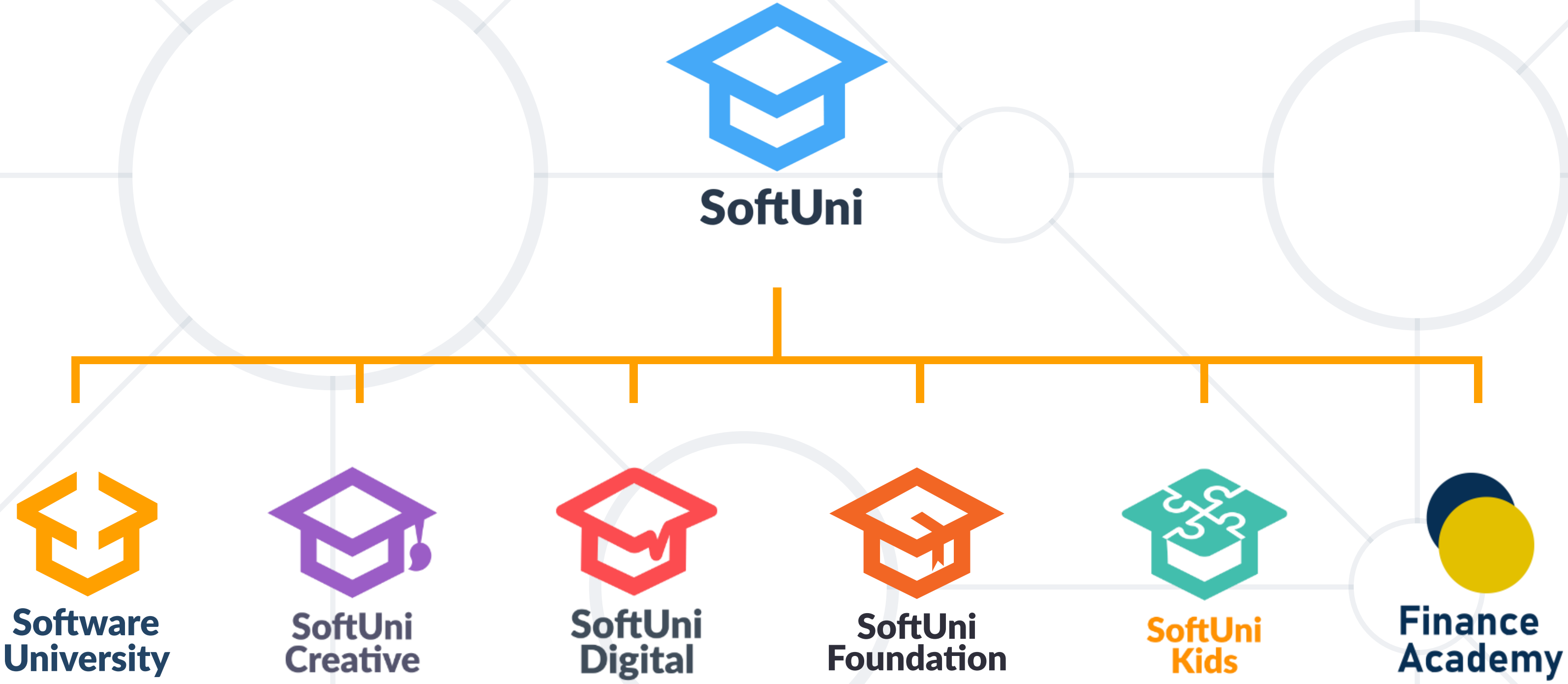
- To **add** new element simply **add new Node** make all the **required references point to it**
- To **remove** Node **clear all the references pointing to it** all the other nodes **should** remain in the the **same order unchanged**

Summary

- **Data structures** organize data in computer systems for efficient use
  - Abstract data types (**ADT**) describe a set of operations
- **Algorithm complexity** is a rough estimation of the **number of steps** performed by given computation
- **Arrays** are a **lightweight data structure** that has **constant time access** to elements but has a **fixed size**

Questions?

Software University



SoftUni Diamond Partners

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter International

INDEAVR Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Software
University

FORUM

License