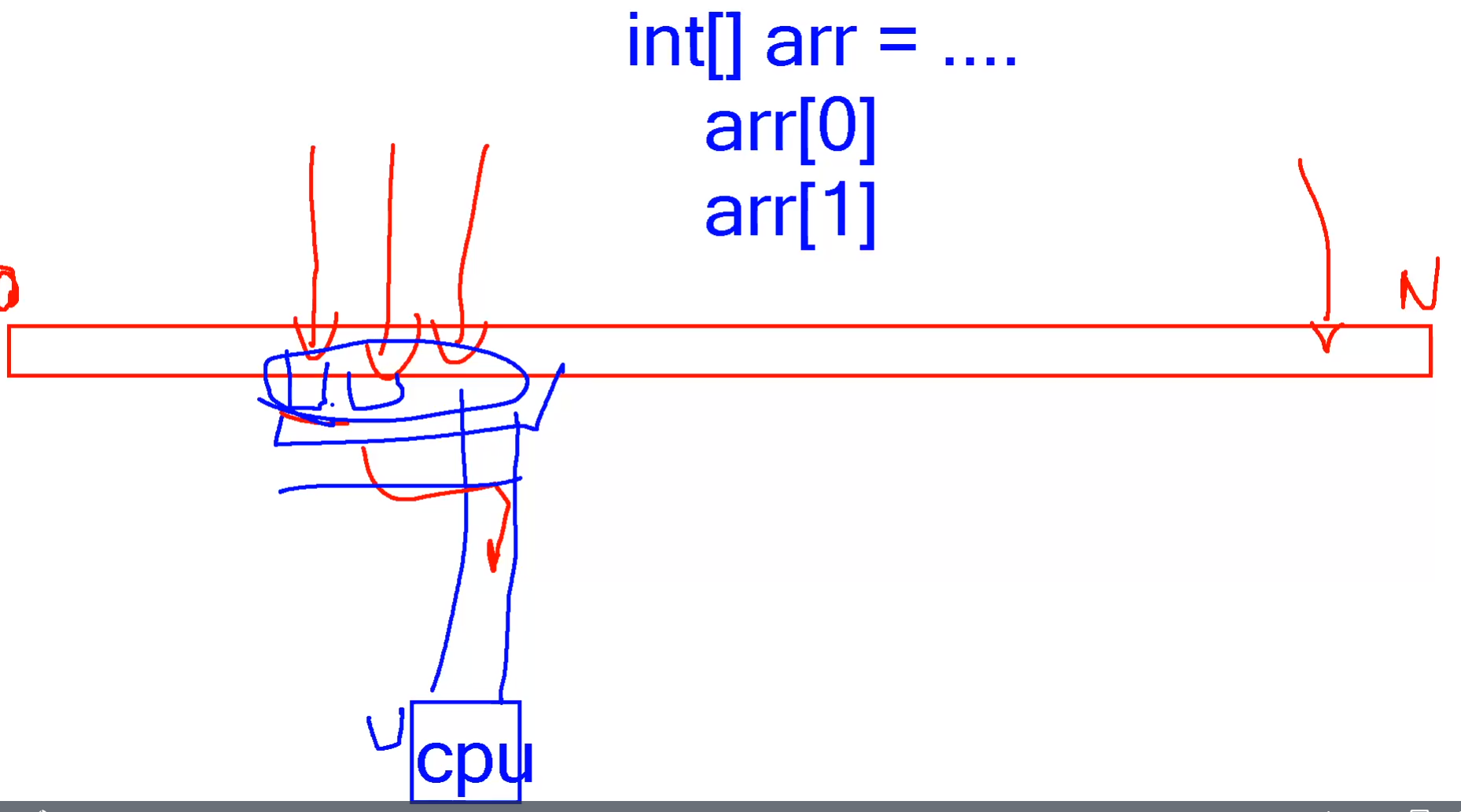
# 1. Data Structures and Complexity

## 1.1. Memory Storage

The term "memory", meaning "primary storage" or "**main memory**", is often associated with adressable **semiconductor(транзистор) memory**.

Много често като искаме единица данни от Рам паметта към процесора, шината заделя исканата единица данни плюс още много поредни следващи единици данни (до размера на шината). Много е вероятно, ако искаме да достъпим първия елемент на масив, то да искаме да достъпим и втория елемент. А втория елемент също ще е минал по шината и ще е вече на най-бързата cache памет на процесора.



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**.

boolen -> 1 byte въпреки че може да е 1 бит (true/false или 1/0) – **защото най-малката адресируема част на паметта е 1 byte**

byte -> 1 byte

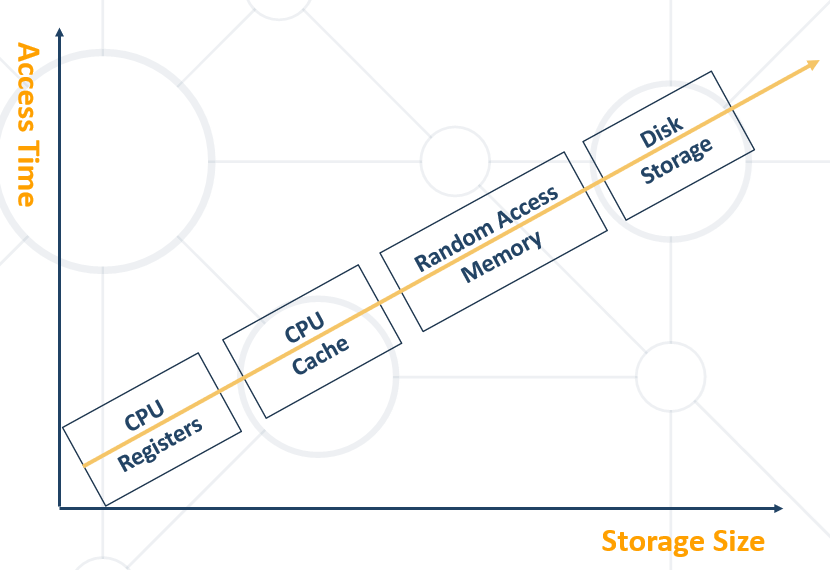
short -> 2 bytes = 16 bits

int -> 4 byte = 32 bits

long ->

### 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**.



## 1.2. 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.**

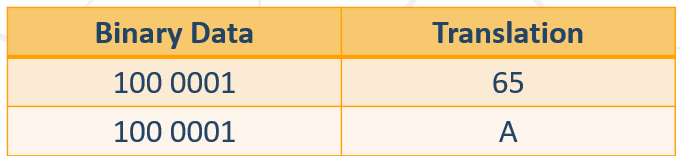
### What is information?

* **"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:

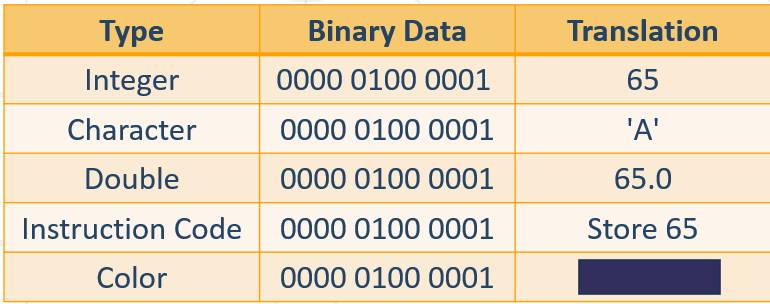
**The history of temperature readings all over the world for the past 100, when organized and analyzed we find that global temperature is rising. – That is information.**

### 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:

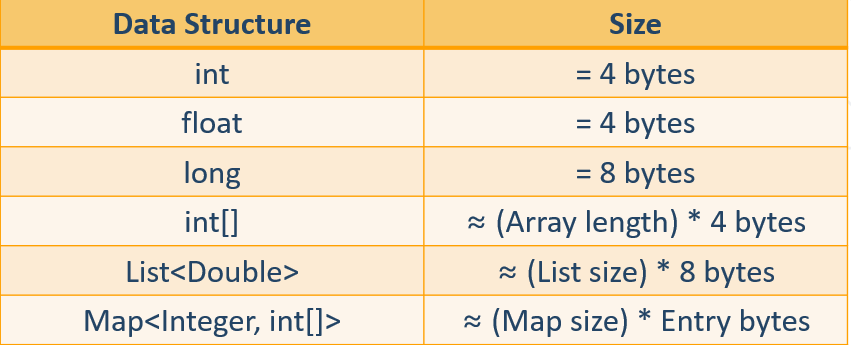


* It is easy to notice, that the way we **read** the data **retrieves the information** of the bits in different ways. However those bits have only **0** or **1** as values.
* Example:



### Data Structures

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



### Abstract Data Structures (ADS)

* An **Abstract Data Structure** **(ADS = ADT type)** – 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**;  
 }  
}

## 1.3. Algorithmic Complexity

### Algorithm Analysis

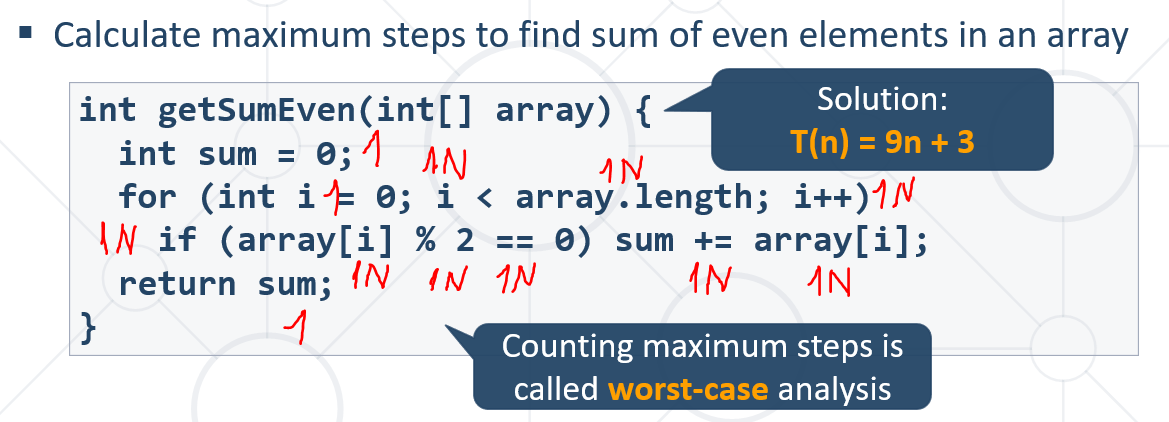
* Why should we analyze algorithms?
  + Predict the **resources** the algorithm will need
    - Computational time (**CPU** consumption)
    - Memory space (**RAM** consumption)
    - Communication **bandwidth** consumption
    - **Hard disk** operations
* There are three main properties we want to analyze:
  + **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)
* 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**

### Consumption of CPU

#### Step Count

Assume that a **single step** is a single CPU instruction.

Гледаме колко стъпки има алгортъма ни, а не колко памет е заета (което е различно)



Инструкция на процесора можем да кажем, че е код завършващ с точка и запетая накрая.

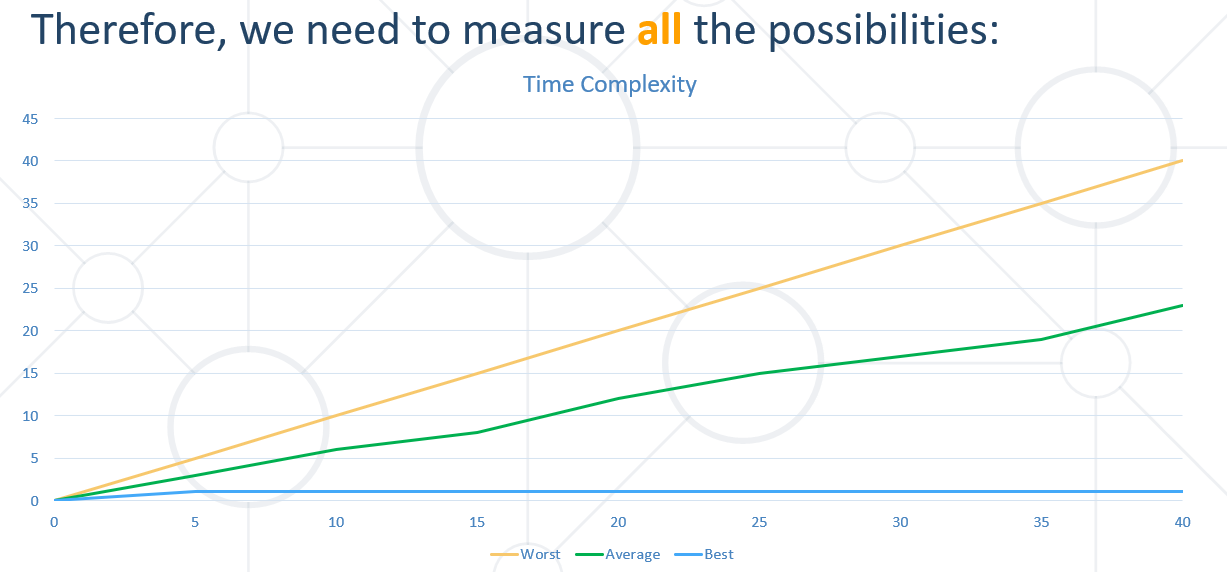
Реално повече инструкции на процесора има на 1 ред код -аритметични/логически и т.н.

#### Simplifying Step Count

* Some parts of the equation **grow much faster** than others
  + T(n) = **3(n2)** + 3n + 3
  + We can **ignore** some part of this equation
  + Higher terms **dominate** lower terms – **n > 2**, **n2 > n**, **n3 > n2**
  + Multiplicative constants can be **omitted** – **12n 🡪 n**, **2n2 🡪 n2**
  + The solution for T(n) = **3(n2)** + 3n + 3 becomes **≈ n2**

#### 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)



* 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:

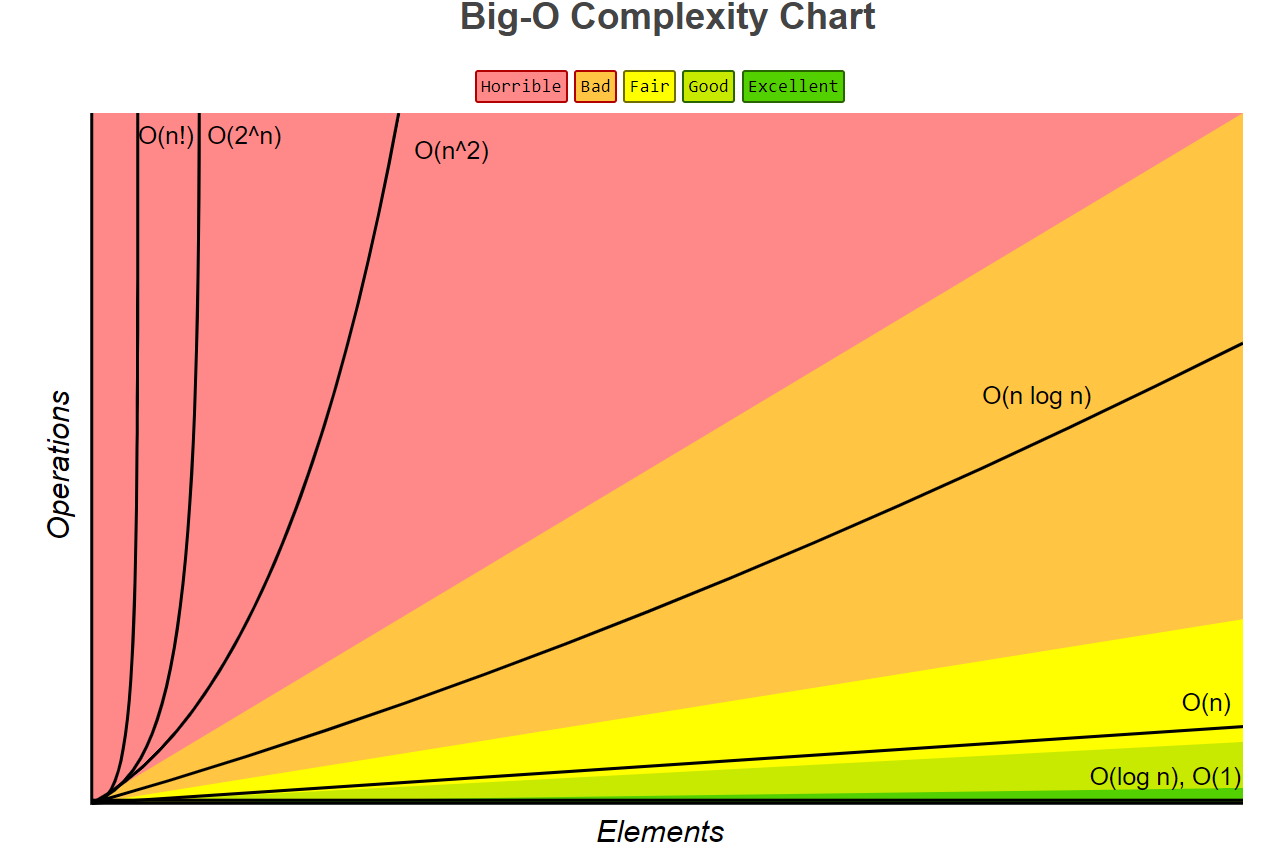
* **Asymptotic notations** are descriptions that allow us to examine an algorithm's running time by expressing its **performance** as the input size, **n**, of an algorithm or a function **f increases**. There are **three** common asymptotic notations:
  + Big **O – O(f(n)) – worst case - in our course**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

**Algorithmic complexity** – rough estimation of the number of steps performed by given computation, depending on the size of the input

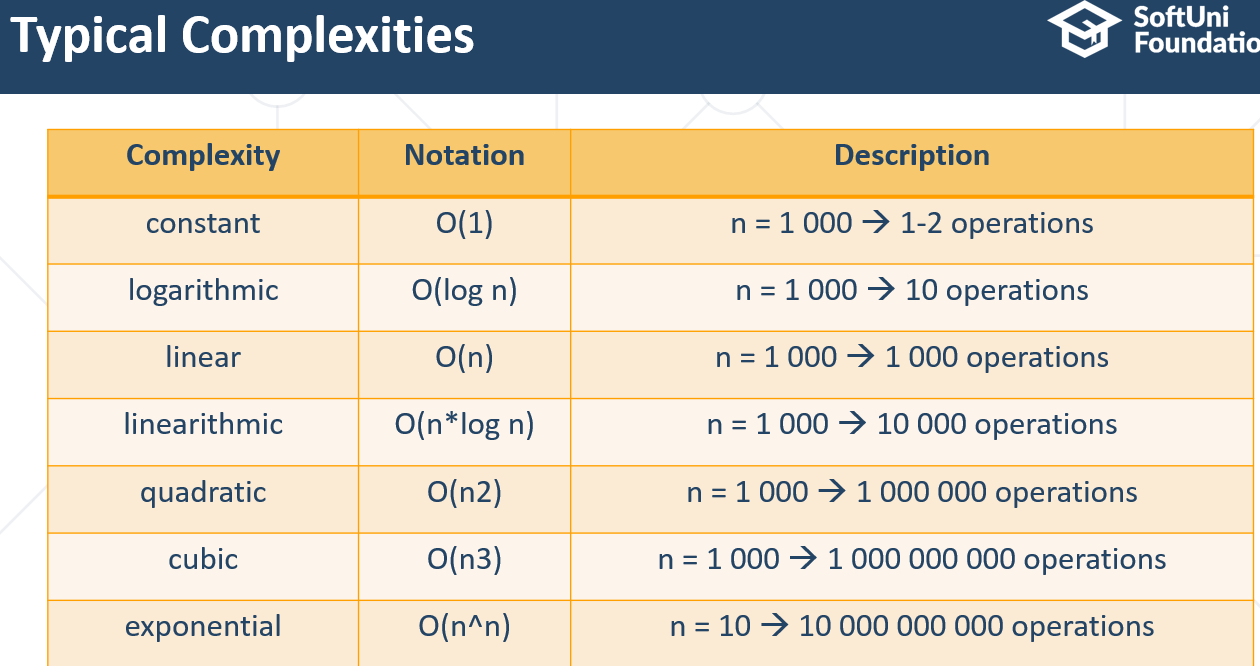
* Measured with asymptotic notation
  + **O(f(n))** – **upper bound (worst case)**
  + **Θ(f(n))** – average case
  + **Ω(f(n))** – lower bound (best case)
    - Where **f(n)** is a function of the size of the input data

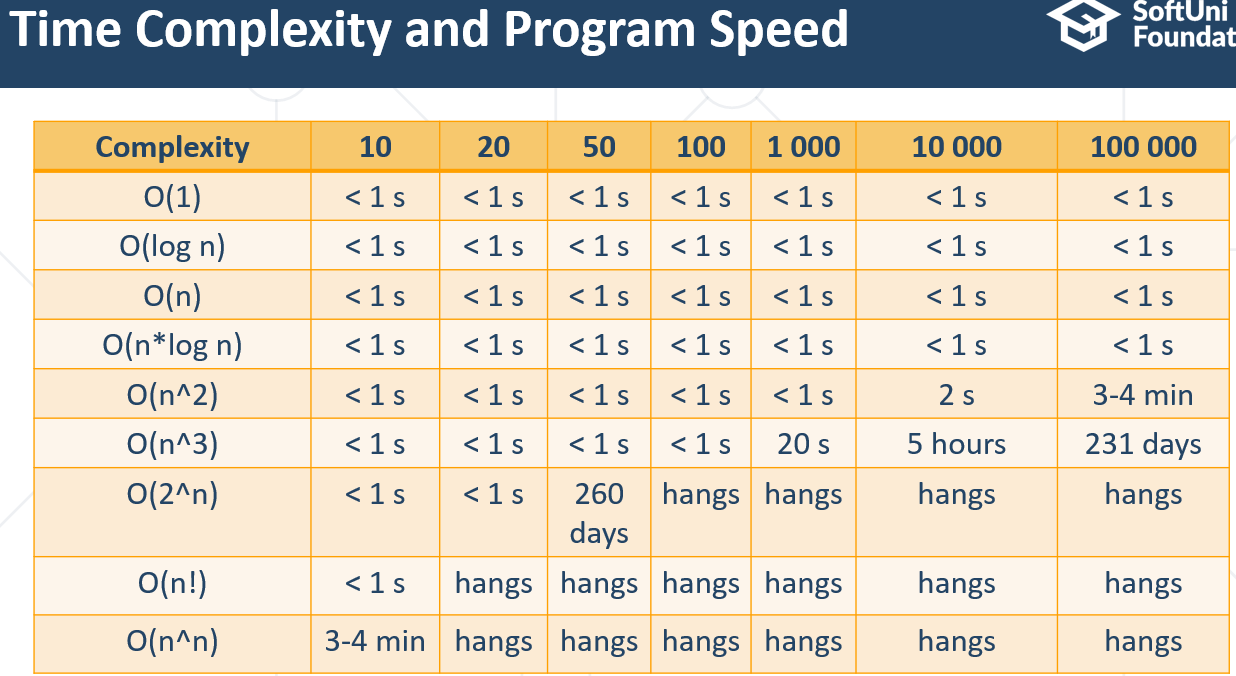
In this course we will analyze only the Big O – **O(f(n)) upper bound (worst case).**

**Ако два алгоритъма имат Big O което е еднакво, то чак тогава може да се наложи да гледаме Theta или Omega.**



* O(1) – Constant time – time does not depend on **N**
* O(log(N)) – Logarithmic time – grows with rate as **log(N)** *log2 64 = 6 (2^6 = 64)*
* O(N) – Linear time grows at the same rate as **N**
* O(N^2),O(N^3) – Quadratic, Cubic grows as square or cube of **N**
* O(2^N) – Exponential grows as **N** becomes the exponent worst algorithmic complexity





### Performance of RAM

* **Memory consumption** should also be considered, for example:
  + Storing elements in a matrix of size N by N
    - Filling the matrix – Running time **O(n2)**
    - Get element by index – Running time **O(1)**
    - Memory requirement **O(n2)**
* However in this course we **won't be optimizing** memory consumption we will only point it out

## 1.4. Array Data Structure

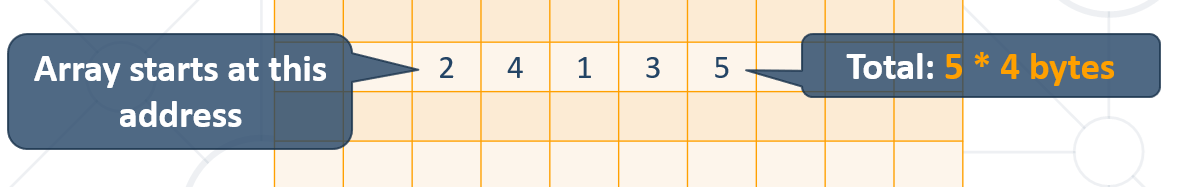
* Ordered
* Very **lightweight**
* Has a **fixed size**
* Usually **built into the language**
* Many collections are implemented by using arrays, e.g.
  + **ArrayList<E>** in Java
  + **ArrayDeque<E>** in Java

### Why Arrays Are Fast?

* Arrays use a **single block of memory = size of the array \* size of the data type**



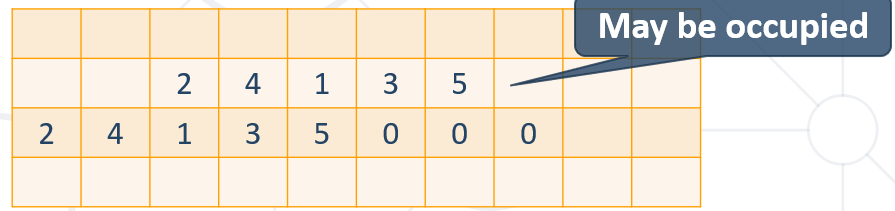
* Uses total of **array pointer + (N \* element/pointer size)**



* **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**

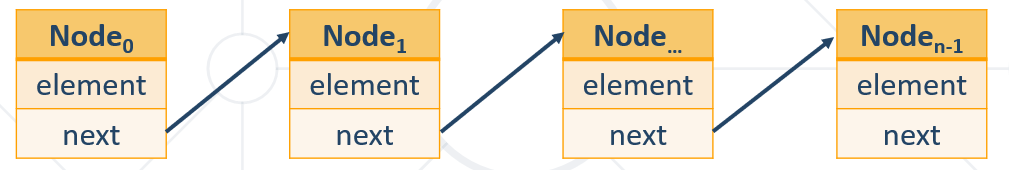


* Array Copy – **O(n)**
* Array Search element – **O(n)**

## 1.5. Data Structure Implementation - Elements Representation Approaches

### How Do We Store the Elements?

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



**public class** ArrayStorage {  
 **private final int INITIAL\_CAPACITY** = 4;  
  
 **private int**[] **elements**;  
 **private int index**;  
  
 **public** ArrayStorage() {  
 **this**.**elements** = **new int**[**INITIAL\_CAPACITY**];  
 **this**.**index** = 0;  
 }  
  
 **public boolean** add(**int** element) {  
 add(element, ++**index**);  
 **return true**;  
 }  
  
 **private void** add(**int** element, **int** index) {  
 **if** (index == **this**.**elements**.**length**) {  
 *//* ***TODO: Add grow method call here***

***private void grow() { - What is the complexity? – O(n)***

***// Create new array with larger size***

***// Copy the elements from the old to the new array***

***// Do additional operations if needed***

***}***

}  
 **this**.**elements**[index] = element;  
 }  
  
 *//* ***TODO: Implement additional operations like: remove(int element), contains(int element) and more***}

**public class** NodeStorage {  
 **private** Node **node**;  
  
 **class** Node {  
 **private int element**;  
 **private** Node **next**;  
  
 Node(**int** element) {  
 **this**.**element** = element;  
 }  
 }  
  
 **public** NodeStorage() {  
 **this**.**node** = **new** Node(0);  
 }  
  
 **public boolean** add(**int** element) {  
 **this**.**node**.**next** = **new** Node(element);  
 **return true**;  
 }  
  
 *//* ***TODO: How do we iterate over the items? How do we remove? How do we iterate and access data?***}

# 2. Linear Data Structures

Static and Dynamic Implementation

## 2.1. Dynamic Arrays – static implementation

* ArrayList is the **implementation** of ADS (Abstract Data structure) **List**
  + Built **atop** **an** **array**, which is able to dynamically **grow** and **shrink** as you **add/remove** elements
* Stores the **elements** inside an array

**public class** ArrayList<E> **implements** List<E> {

**private** Object[] **elements**;

}

Supported operations and complexity:

* + **size()**, **isEmpty()**, **get()**, **set()** – **O(1)**
  + **add()** – the operation runs in **amortized constant** time – повечето пъти времето е константно(рядко се преузмерява)
  + adding **n** elements requires **O(n)** time
  + all of the other operations like: **add(int index, E element)**, **contains()**, **indexOf(), remove(int index) etc**., run in **linear** **time** **O(n)** (roughly speaking)

### ArrayList – Add O(n)

* Implemented **using an array**
* Adding **new** **item** requires **new** **array**



* This approach will copy **all the elements** for each add operation – **O(n)**

### ArrayList – Add O(1)

* Implemented **using an array**
* When **adding**, if needed **double** the size



* This approach will copy at **log(n)** 🡪 n = 109, only ~33 copies – **O(1) amortized**

### Constructor and fields:

**public class** ArrayList<E> **implements** List<E> {  
 **private static final int *DEFAULT\_CAPACITY*** = 4;  
 **private** Object[] **elements**;  
 **private int size**;  
  
 **public** ArrayList() {

**this**.**elements** = **new** Object[***DEFAULT\_CAPACITY***];

}  
}

### Add - Adds an element after the last element:

**public boolean** add(E element) {  
 **if**(**this**.size == **this**.elements.length) {  
 **this**.elements = grow();  
 }  
   
 **this**.elements[**this**.size++] = element;  
   
 **return true**;  
}

### Get - Returns an element at index:

**public** E get(**int** index) {  
 checkIndex(index);  
 **return this**.getElement(index);  
}  
  
**private** E getElement(**int** index) {  
 **return** (E) **this**.elements[index];  
}

**private void** checkIndex(**int** index) {  
 **if** (index < 0 || index >= **size**) {  
 **throw new** IllegalArgumentException();  
 }  
}

### Set - Sets an element at index:

**public** E set(**int** index, E element) {  
 checkIndex(index);  
 E oldElement = **this**.getElement(index);  
 **this**.elements[index] = element;  
 **return** oldElement;  
}

### Remove - Removes and returns an element at index:

**public** E remove(**int** index) {  
 **this**.checkIndex(index);  
 E element = **this**.getElement(index);  
 **this**.elements[index] = **null**;  
 **this**.size--;  
 shift(index);  
 ensureCapacity();  
  
 **return** element;  
}

### Grow and Shrink

**private** Object[] grow() {  
 **return** Arrays.copyOf(**this**.elements, **this**.elements.length \* 2);  
}

*//increase the size of the list***if** (**size** >= **elements**.**length**) {  
 Object[] newElements = **new** Object[**size** \* 2];  
 **for** (**int** i = 0; i <**elements**.**length**; i++) {  
 newElements[i] = **elements**[i];  
 }  
  
 **elements** = newElements;  
}

**private** Object[] shrink() {  
 **return** Arrays.copyOf(**this**.elements, **this**.elements.length / 2);  
}

## 2.2. Nodes - Building Block

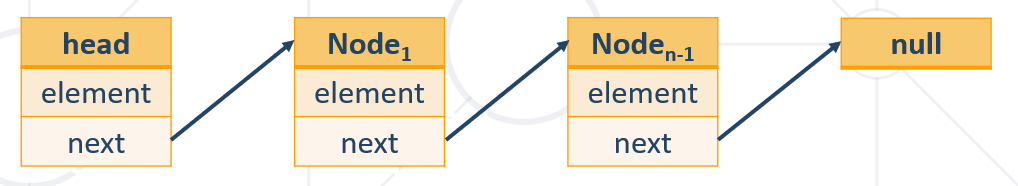
### Node Class

* The **Node** class is the **build block** for many data structures
* Inside Node object we store **an element and pointer to the next node at least**
* However, we **can** **store anything else**

**private static class** Node<E> {  
 **private** E **element**; *// Must have* **private** Node<E> **next**; *// Must have* **private** Node<E> **previous**; *// Additional*}

* Many data structures use **node chaining**

**public class** LinkedList<E> **implements** Deque<E> {  
 **private** Node<E> **head**;  
}



### SinglyLinkedList – dynamic implementation

* Linear data structure where each **element** is a **separate object** – **Node**
* The elements are **not** storedat **contiguous** memory
* The entry point is commonly the **head** of the list

However we define what is the entry point

**public class** DoublyLinkedList<E> **implements** LinkedList<E> {  
 **private** Node<E> **head**;  
 **private int size**;

* Supported operations and complexity:
  + **addFirst**(), **removeFirst**(), **getFirst**(), **size**()– **O(1)**
  + How about operations on the **last element**?
    - **addLast**(), **removeLast**(), **getLast**() – again depends if we keep the reference to the last node or no can be constant – **O(1)** or linear – **O(n)**
  + operations that **index** into the list will run in **linear** **time** **O(n)** (roughly speaking)

### DoublyLinkedList – dynamic implementation

However we define what is the entry point

**public class** DoublyLinkedList<E> **implements** LinkedList<E> {  
 **private** Node<E> **head**;  
 **private** Node<E> **tail**;  
 **private int size**;

**public** Node<E> getHead() {  
 **return head**;  
}  
  
**public static class** Node<E> {  
 **private** E **element**; *// Must have* **private** Node<E> **next**; *// Must have* **private** Node<E> **previous**; *// Additional* **public** Node(E element, Node<E> next, Node<E> previous) {  
 **this**.**element** = element;  
 **this**.**next** = next;  
 **this**.**previous** = previous;  
 }  
  
 **public** E getElement() {  
 **return element**;  
 }  
  
 **public** Node<E> getNext() {  
 **return next**;  
 }  
  
 **public** Node<E> getPrevious() {  
 **return previous**;  
 }  
  
 **public** Node setNext(Node<E> next) {  
 **this**.**next** = next;  
 **return this**;  
 }  
  
 **public** Node setPrevious(Node<E> previous) {  
 **this**.**previous** = previous;  
 **return this**;  
 }  
}

хитро и работи 😊

@Override  
**public void** addLast(E element) {  
 Node<E> newNnode = **new** Node<>(element, **null**, **tail**);  
 **if** (**head** == **null**) {  
 **head** = newNnode;  
 }  
  
 **if** (**tail** != **null**) {  
 **tail**.setNext(newNnode);  
 }  
  
 **tail** = newNnode;  
  
 **size**++;  
}

хитро и работи 😊

**public void** remove(Node<E> node) {  
 **if** (**head** == node) {  
 **head** = node.getNext();  
 }  
  
 **if** (**tail** == node) {  
 **tail** = node.getPrevious();  
 }  
  
 **if** (node.getPrevious() != **null**) {  
 node.getPrevious().setNext(node.getNext());  
 }  
  
 **if** (node.getNext() != **null**) {  
 node.getNext().setPrevious(node.getPrevious());  
 }  
   
 **size**--;  
}

### Built-in Class LinkedList in JAVA – doubly-linked queue with static implementation for quick performance

**public class** LinkedList<E>  
 **extends** AbstractSequentialList<E>  
 **implements** List<E>, Deque<E>, Cloneable, java.io.Serializable

**public interface** List<E> **extends** Collection<E> {

**public interface** Collection<E> **extends Iterable<E>** {

LinkedList<String> builtInLinkedList = **new** LinkedList<>();  
  
DoublyLinkedList<String> people = **new** DoublyLinkedList<>();  
builtInLinkedList.addLast(**"joro"**);  
builtInLinkedList.addLast(**"pesho"**);  
builtInLinkedList.addLast(**"misho"**);  
builtInLinkedList.addLast(**"grisho"**);

**Обхождане с iter (foreach)**  
**for** (String lrr : builtInLinkedList) {  
 System.***out***.println(lrr);  
}

**Обхождане с iterator**  
Iterator<String> iterator = builtInLinkedList.iterator();  
**while** (iterator.hasNext()){  
 String person = iterator.next(); *//връща стринг* System.***out***.println(person);  
}

**Премахване на елемент – работи и с ArrayList<String> тъй като iterator работи с всяка колекция, която имплементира Iterable интерфейса**

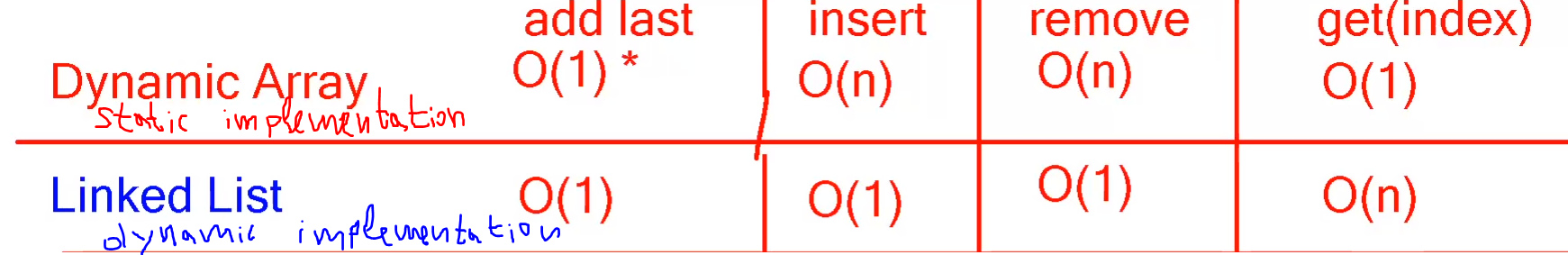
**while** (iterator.hasNext()) {  
 String person = iterator.next(); *//връща стринг* **if** (person.equals(**"pesho"**)) {  
 iterator.**remove();**  
 } **else** {  
 System.***out***.println(person);  
 }  
}

## 2.3. Сравнение на статична и динамична имплементация

При статичната имплементация, при вмъкване или изтриване на елемент, пренареждаме всички останали, но пък имаме директен достъп до елементите.

При динамичната имплементация, при вмъкване или изтриване на елемент става веднага, но пък имаме обхождане през next / previous докато стигнем до даден елемент.

What is the time complexity to search (not to select a concrete) an element in an array (statis implemenation)? - **O(n)**



Time complexity of a hashset when contains operation - O(1) which is very clever implementation of a hashset.

<https://stackoverflow.com/questions/6574916/hashset-look-up-complexity>

## 2.4. Stacks - dynamic implementation

* Stack is the **implementation** of ADS **LIFO L**ast **I**n **F**irst **O**ut
  + Build by using **Node** class or atop an **array**
* Stack example using Node

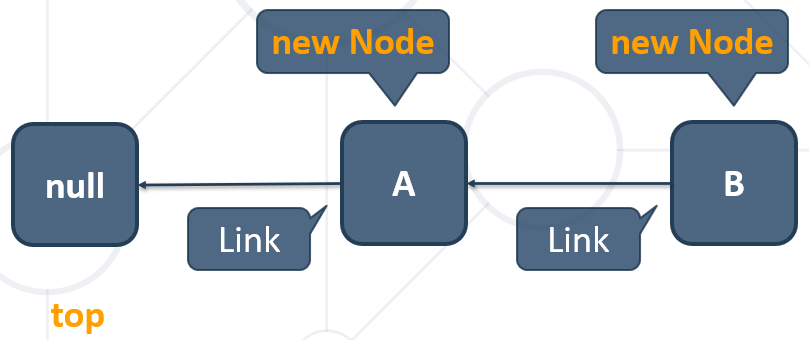
**public class** Stack<E> **implements** AbstractStack<E> {  
 **private** Node<E> **top**;  
 **private int size**;  
}

* Supported operations and complexity:
  + **size**(), **isEmplty**(), **push**(), **pop**(), **peek**()– **O(1)**
  + all of the other operations run in linear time (roughly speaking):
    - **forEach**()
    - **contains**()

etc…

### Stack – Push

* Chain the nodes by using the **top** field:

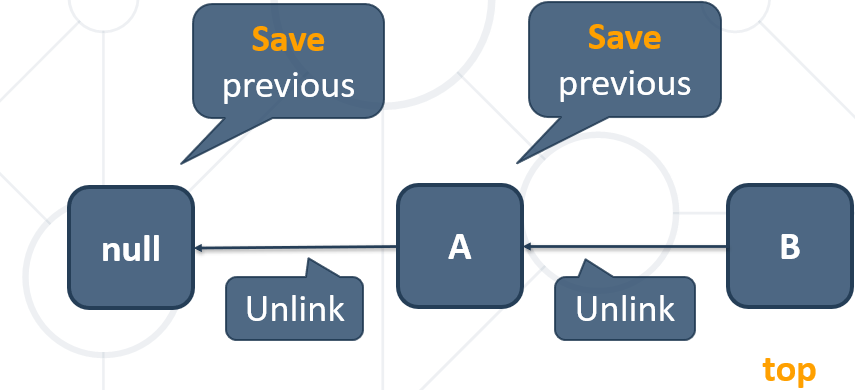


* Add element at the top
  + **Link** the nodes and **increment** size

**public void** push (E element){  
 Node<E> newNode = **new** Node<>(element);  
 newNode.previous = top;  
 top = newNode;  
 **this**.size++;  
}

### Stack – Pop

* Remove the **top** Node and return the element
  + **Unlink** the nodes and **decrease** size



* Remove and return element at the top:

**public** E pop () {  
 ensureNonEmpty();  
 E element = **this**.top.element;  
 Node<E> temp = **this**.top.previous;  
 **this**.top.previous = **null**;  
 **this**.top = temp;  
 **this**.size--;  
 **return** element;  
}

## 2.5. Queues – dynamic implementation

* Queue is the **implementation** of ADS **FIFO F**irst **I**n **F**irst **O**ut
  + Build by using **Node** class or atop an **array**
* Queue example using Node

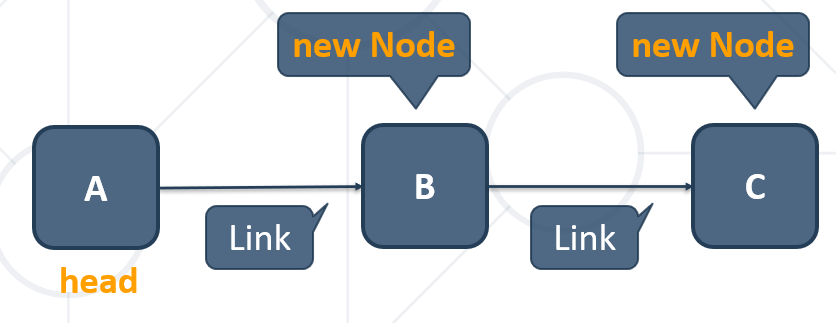
**public class** Queue<E> **implements** AbstractQueue<E> {  
 **private** Node<E> **head**;  
 **private int size**;  
}

* Supported operations and complexity:
  + **size**(), **isEmplty**(), **poll**(), **peek**()– **O(1)**
  + **offer**():
    - if we keep the reference to the that node – **O(1)**
    - If we have to chase pointers to that node – **O(n)**
  + all of the other operations run in linear time (roughly speaking):

**forEach**(), **contains**(),etc…

**Queue – Offer**

* Head **==** null 🡪 head **=** **new Node**
* Size **>** 0 🡪 chain the nodes by adding **new Node** after the last one the so called **tail**:



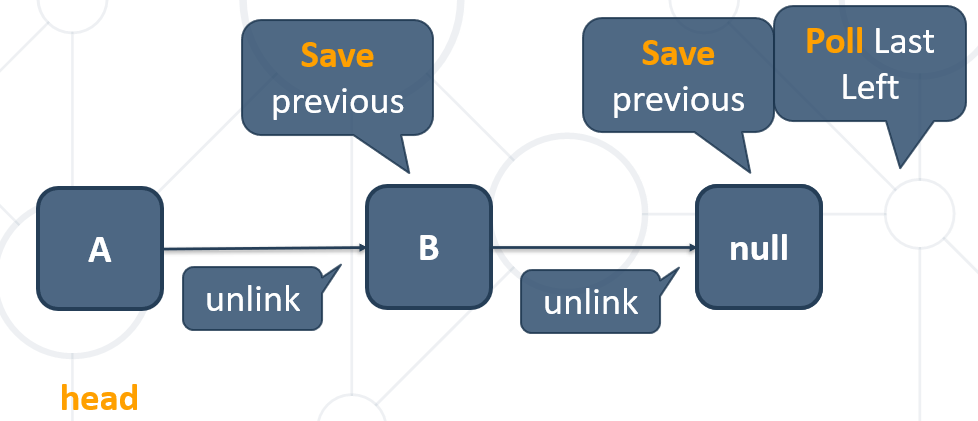
### Queue – Offer

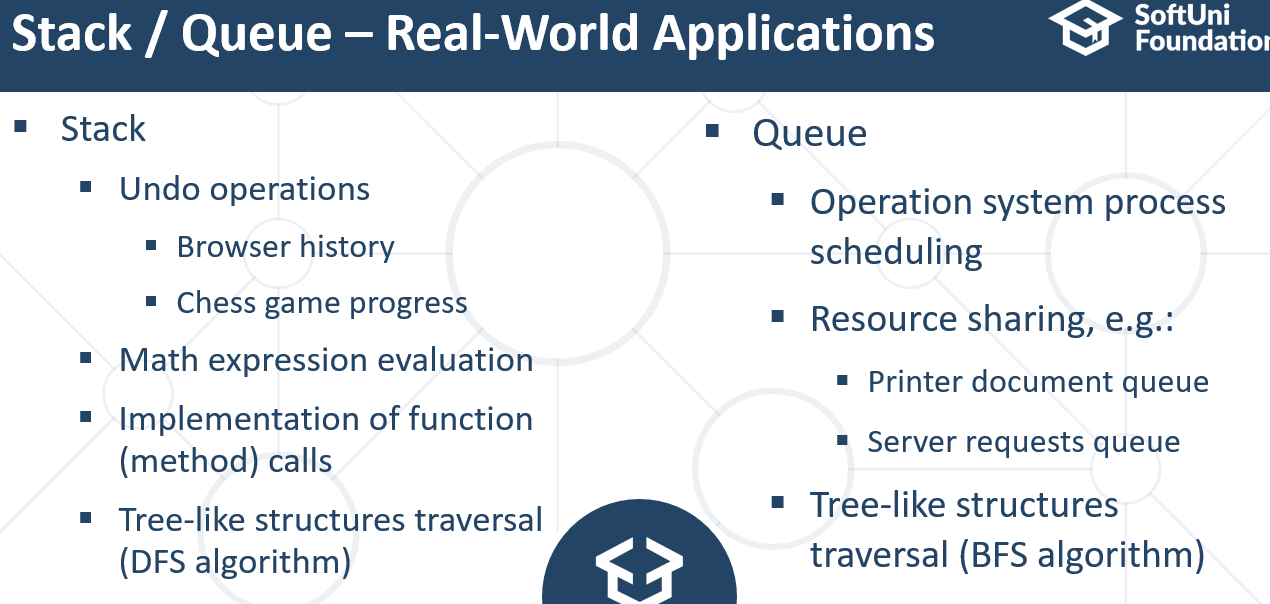
* Add element at the end – **Link** the nodes and **increase** size

**public void** offer (E element){  
 Node<E> newNode = **new** Node<>(element);  
 **if** (**this**.head == **null**) {  
 **this**.head = newNode;  
 } **else** {  
 Node<E> current = **this**.head;  
 **while** (current.next != **null**) {  
 current = current.next;  
 }  
 current.next = newNode;  
 }  
 **this**.size++;  
}

### Queue – Poll

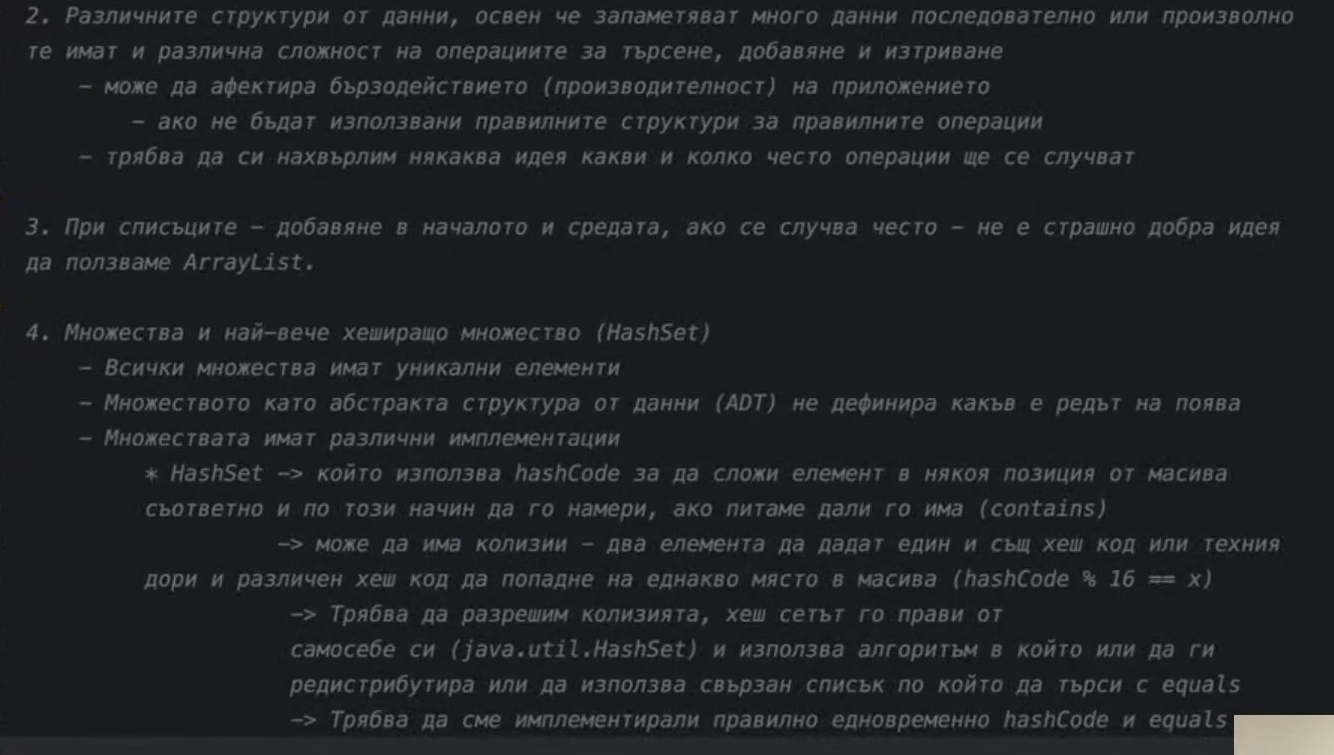
* Remove the **head** Node and return the element
  + **Unlink** the node and **decrease** size





## 2.6. Още някои сравнения

При **Hash**set и **Hash**map



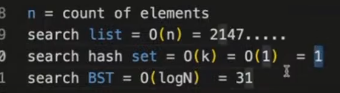
Ако модулното делене на примерно 16 за различни хешкодове даде един и същи резултат/остатък, тогава влиза в действие **equals** функцията.

**HashSet and TreeSet**



**BST, Set and list**

Как се получават броя стъпки при BinarySearchTree когато търсим да са логаритъм от N - като разделяме на 2 и за Integer(2 147 \*\*\* \*\*\*) реално са 31 стъпки.



И самото вкарване на елемент изисква ребалансиране на BinarySearchTree дървото - хубавото е, че вкарването на нов елемент изисква същия малък брой стъпки, а именно основа 2 log N или два-три пъти на брой операции основа 2 log N - което е пренебрежимо малко в сравнение обхождането на LinkedList примерно

# 3. Trees Representation and Traversal (BFS, DFS)

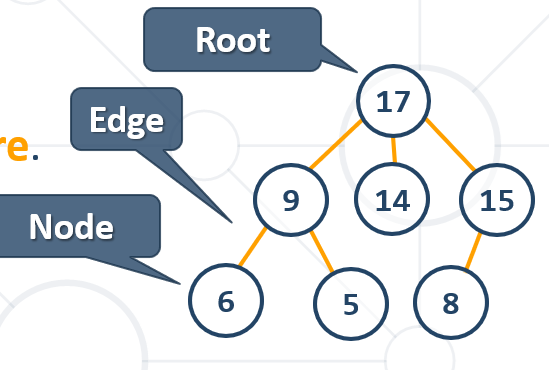
## 3.1. Trees and Related Terminology

**Tree Definition**

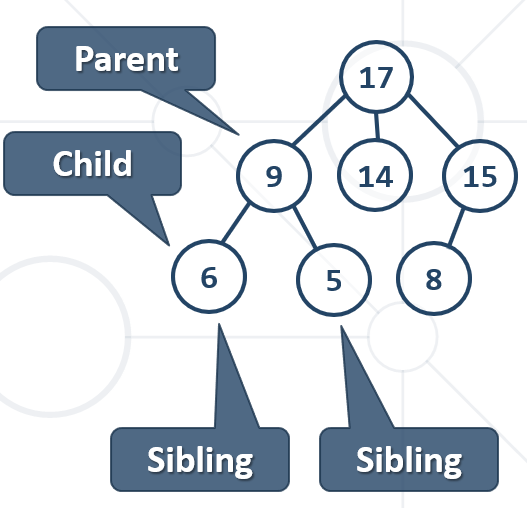
* Tree is a widely used **abstract data type** (ADT) that simulates a hierarchical **tree structure**, with a root value and subtrees of children with a **parent node**, represented as a set of linked **nodes**.
* **Recursive definition** – a tree consists of a value and a forest (the subtrees of its children)
* One **reference** can point to **any** **given** **node** (a node has at **most** a **single** parent), and **no** **node** in the **tree** **point to the root**. Every node (other than the root) **must** have exactly **one** **parent**, and the **root** **must** have **no** **parents**.

**Tree Data Structure – Terminology**

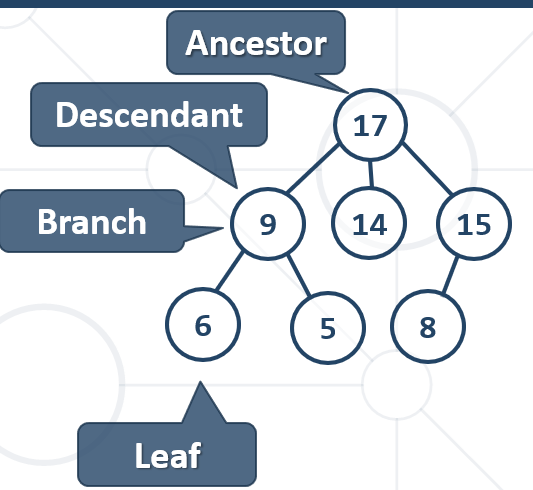
* **Node (Връх)** – a structure which may contain a **value** or condition, or represent a separate **data** **structure**.
* **Edge (Ребра)** – the **connection** **between** one **node** and **another**.
* **Root (Корен)** – the **top** node in a **tree**, the **prime** **ancestor**.



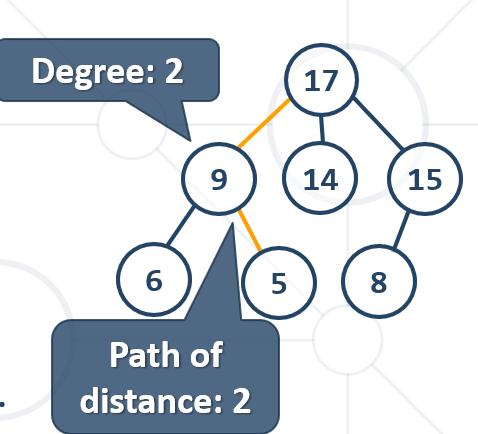
* **Parent (Родител)** – the **converse** notion of a **child**, an **immediate** **ancestor (прародител)**.
* **Child (Дете)** – node **directly** connected to **another** node when moving **away** from the **root**, an immediate descendant.
* **Siblings (Близнаци, братя, сестри, които имат общ родител)** – a **group** of **nodes** with the **same** **parent**.



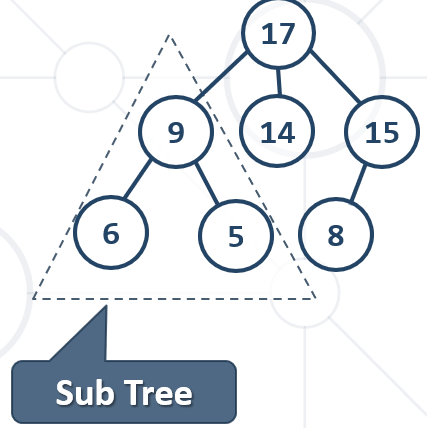
* **Ancestor(Предшественик)** – node reachable by repeated proceeding **from** **child** **to** **parent**.
* **Descendant (Потомък)** – node reachable by repeated proceeding **from parent** **to** **child**.
* **Leaf (Листо)** – node with **no** **children**.
* **Branch (Клон)** – node with **at least one child**.



* **Degree (Степен)** – number of children for node zero for a leaf.
* **Path** – sequence of nodes and edges connecting a node with a descendant.
* **Distance** – number of edges along the shortest path between two nodes.
* **Depth** – distance between a node and the root.



* **Level** – depth + 1.
* **Height** – The number of edges on the longest path between a node and a descendant leaf. = **Height** – the maximum level in the tree.
* **Width** – number of nodes in a level.
* **Breadth** – number of leaves.
* **Forest** – set of disjoint trees.
  + {17}, {9, 6, 5}, {14}, {15, 8}
* **Sub Tree** – tree T is a tree consisting of a node in T and all of its descendants in T.



### ВАЖНО:

**Дървовидната структура е вид граф когато от root-a (корена) мога да стигна до всяко едно място (връх или листо) само по един път/ по един начин.**

**Докато при graph (граф) – неща, които са свързани помежду си без никакви изисквания**

## 3.2. Implementing Trees - Recursive Tree Data Structure

* The recursive definition for **tree** data structure:
  + A single node **is a tree**
  + Nodes have **zero or multiple children** that are **also trees**

**public class** Tree<E> **implements** AbstractTree<E> {  
 **private** E **key**;  
 **private** Tree<E> **parent**;  
 **private** List<Tree<E>> **children**;  
  
 **public** Tree(E key, Tree<E>... children) {  
 **this**.**key** = key;  
 **this**.**children** = **new** ArrayList<>();  
 **for** (Tree<E> child : children) {  
 **this**.**children**.add(child);  
 child.**parent** = **this**;  
 }  
 }

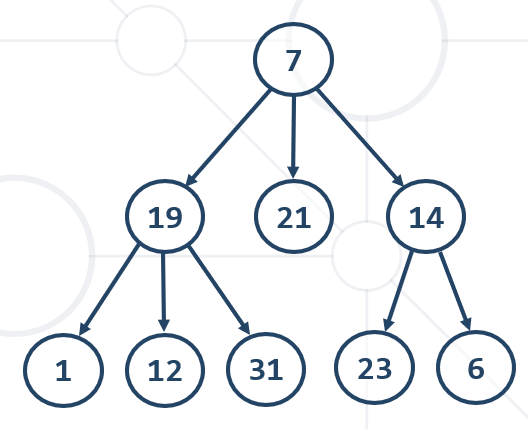
}

**Или използваме името Node вместо Tree**

**public class** Node<E> **implements** NodeTree<E> {  
 **private** E **key**;  
 **private** Node<E> **parent**;  
 **private** List<Node<E>> **children**;  
  
 **public** Tree(E key, Node<E>... children) {  
 **this**.**key** = key;  
 **this**.**children** = **new** ArrayList<>();  
 **for** (Node<E> child : children) {  
 **this**.**children**.add(child);  
 child.**parent** = **this**;  
 }  
 }

}

**public class** Main {  
 **public static void** main(String[] args) {  
 Tree<Integer> tree =  
 **new** Tree<>(7,  
 **new** Tree<>(19,  
 **new** Tree<>(1),  
 **new** Tree<>(12),  
 **new** Tree<>(31)),  
 **new** Tree<>(21),  
 **new** Tree<>(14,  
 **new** Tree<>(23),  
 **new** Tree<Integer>(6))  
 );  
 }  
}



## 3.3. Traversing Tree-Like Structures – Обхождане на дървета

**Traversing a tree** means to visit each of its nodes exactly once

The **order of visiting nodes** may vary on the traversal algorithm

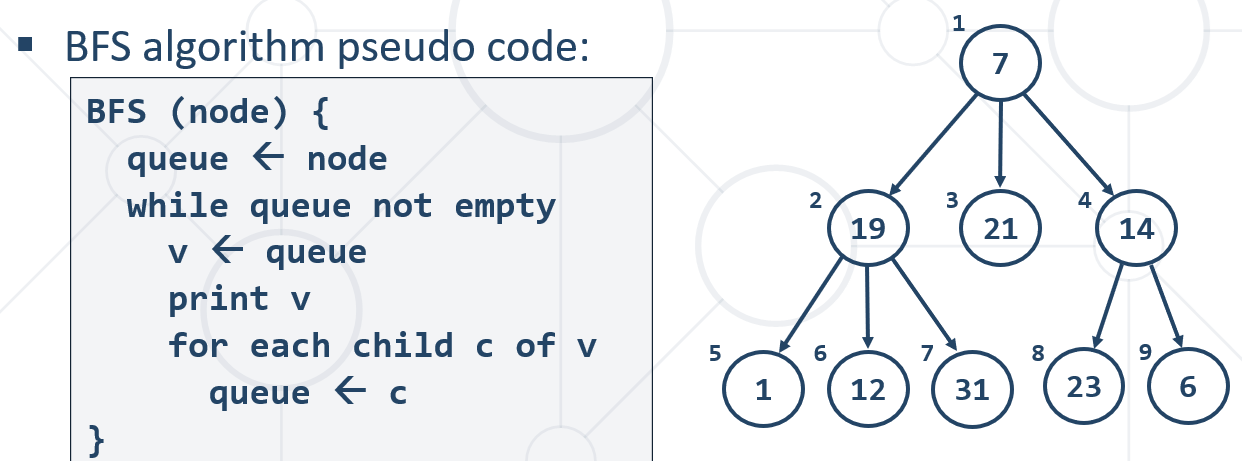
* + **Breadth-First Search** (BFS)
    - Nearest nodes visited first
    - Implemented by a queue and a while loop (recursion also possble)
  + **Depth-First Search** (DFS)
    - Visit node's successors first
    - Usually implemented by recursion (or implemented by a stack and a while loop)

### Breadth-First Search (BFS)

* **Breadth-First Search** (BFS) first visits the neighbor nodes, then the neighbors of neighbors, etc.

#### Имплементация с цикъл и опашка

**7 19 21 14 1 12 31 23 6**



**public** List<E> orderBfs() {  
 List<E> result = **new** ArrayList<>();  
 Deque<Tree<E>> queue = **new** ArrayDeque<>();  
 queue.offer(**this**);

**while** (queue.size() > 0) {  
 Tree<E> current = queue.poll();  
 result.add(current.**key**)

**for** (Tree<E> child : current.**children**)  
 queue.offer(child);  
 }  
 **return** result;  
}

#### Имплементация с рекурсия

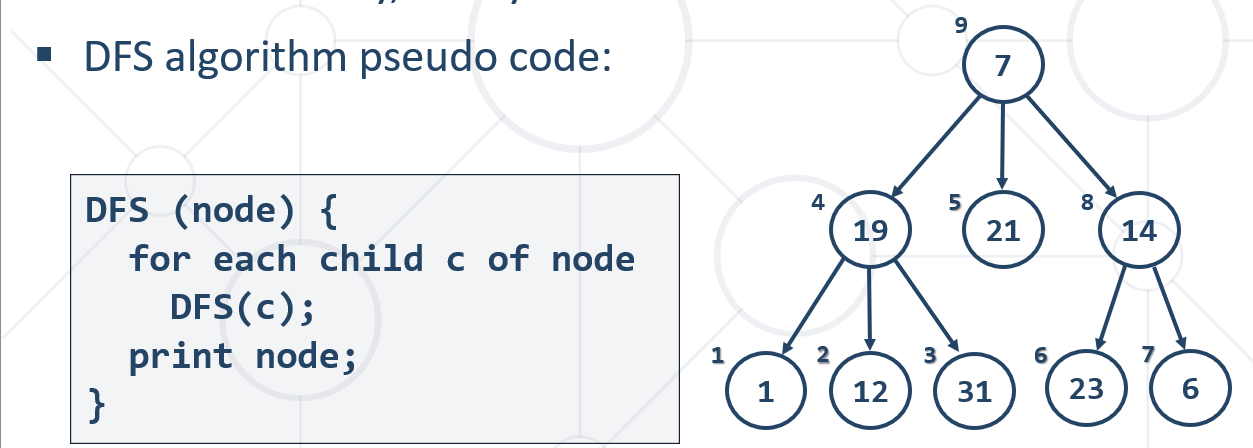
**Не се получава лесно**

### Depth-First Search (DFS)

* **Depth-First Search (DFS)** first visits all descendants of given node recursively, finally visits the node itself
* DFS algorithm pseudo code:

#### Имплементация с рекурсия

**1 12 31 19 21 23 6 14 7**



@Override  
**public** List<E> orderDfs() {  
 List<E> order = **new** ArrayList<>();  
 **this**.dfs(**this**, order);  
 **return** order;  
}  
  
**private void** dfs(Tree<E> tree, List<E> order) {  
 **for** (Tree<E> child : tree.**children**) {  
 **this**.dfs(child, order);  
 }  
 order.add(tree.**key**);  
}

#### Имплементация с цикъл и стек - (в случая тръгва от последния в дълбочина)

**7 14 6 23 21 19 31 12 1**

@Override  
**public** List<E> orderDfs() {  
 List<E> result = **new** ArrayList<>();  
 Deque<Tree<E>> stack = **new** ArrayDeque<>();  
 stack.push(**this**);  
  
 **while** (stack.size() > 0) {  
 Tree<E> current = stack.pop();  
 result.add(current.**key**);  
  
 **for** (Tree<E> child : current.**children**)  
 stack.push(child);  
 }  
  
 **return** result;  
}

## 3.4. Инициализиране на дървото

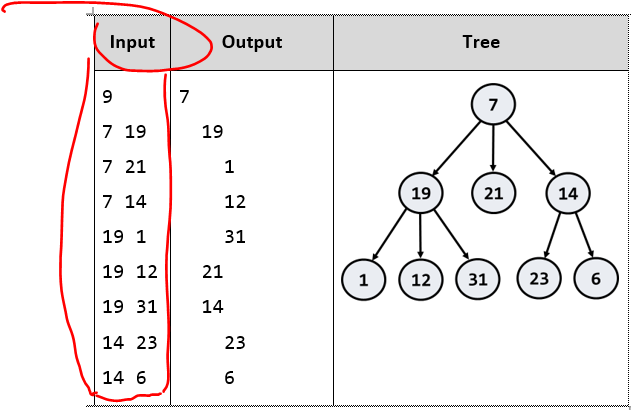
**Вариант 1)**

**public class** Tree<E> **implements** AbstractTree<E> {  
 **private** E **key**;  
 **private** Tree<E> **parent**;  
 **private** List<Tree<E>> **children**;  
  
 **public** Tree(E key, Tree<E>... children) {  
 **this**.**key** = key;  
 **this**.**children** = **new** ArrayList<>();  
 **for** (Tree<E> child : children) {  
 **this**.**children**.add(child);  
 child.**parent** = **this**;  
 }  
 }

}

**public class** Main {  
 **public static void** main(String[] args) {  
 Tree<Integer> tree =  
 **new** Tree<>(7,  
 **new** Tree<>(19,  
 **new** Tree<>(1),  
 **new** Tree<>(12),  
 **new** Tree<>(31)),  
 **new** Tree<>(21),  
 **new** Tree<>(14,  
 **new** Tree<>(23),  
 **new** Tree<Integer>(6))  
 );  
 }  
}

**Вариант 2) – чрез TreeFactory наш клас когато четем поредни двойки родител – дете**



**public class** TreeFactory {  
 **private** Map<Integer, Tree<Integer>> **nodesByKeys**;  
  
 **public** TreeFactory() {  
 **this**.**nodesByKeys** = **new** LinkedHashMap<>();  
 }  
  
 **public** Tree<Integer> createTreeFromStrings(String[] input) {  
 **for** (String line : input) {  
 **int**[] nodeValues = Arrays.*stream*(line.split(**"\\s+"**))  
 .mapToInt(Integer::*parseInt*)  
 .toArray();  
 addEdge(nodeValues[0], nodeValues[1]);  
 }  
 **return** getRoot();  
  
 }  
  
 **private** Tree<Integer> getRoot() {  
 **for** (Tree<Integer> node : **this**.**nodesByKeys**.values()) {  
 **if** (node.getParent() == **null**) {  
 **return** node;  
 }  
 }  
 **return null**;  
 }  
  
 **public** Tree<Integer> createNodeByKey(**int** key) {  
 **this**.**nodesByKeys**.putIfAbsent(key, **new** Tree<Integer>(key));  
 **return this**.**nodesByKeys**.get(key);  
 }  
  
 **public void** addEdge(**int** parent, **int** child) {  
 Tree<Integer> parentTree = **this**.createNodeByKey(parent);  
 Tree<Integer> childTree = **this**.createNodeByKey(child);  
  
 parentTree.addChild(childTree);  
 childTree.setParent(parentTree);  
 }  
}

**public class** Tree<E> **implements** AbstractTree<E> {  
 **private** E **value**;  
 **private** Tree<E> **parent**;  
 **private** List<Tree<E>> **children**;  
  
 **public** Tree(E value, Tree<E>... children) {  
 **this**.**value** = value;  
 **this**.**children** = **this**.initChildren(children);  
 }  
  
 **private** List<Tree<E>> initChildren(Tree<E>[] children) {  
 List<Tree<E>> treeChildren = **new** ArrayList<>();  
  
 **for** (Tree<E> child : children) {  
 child.setParent(**this**);  
 treeChildren.add(child);  
 }  
 **return** treeChildren;  
 }  
  
 @Override  
 **public void** setParent(Tree<E> parent) {  
 **this**.**parent** = parent;  
 }  
  
 @Override  
 **public void** addChild(Tree<E> child) {  
 **this**.**children**.add(child);  
 }  
  
 @Override  
 **public** Tree<E> getParent() {  
 **return this**.**parent**;  
 }  
  
 @Override  
 **public** E getKey() {  
 **return this**.**value**;  
 }  
  
 **public** List<Tree<E>> getChildren() {  
 **return this**.**children**;  
 }

}

В main-а:

String[] input = {  
 **"19 1"**,  
 **"19 12"**,  
 **"19 31"**,  
 **"14 23"**,  
 **"14 6"**,  
 **"7 19"**,  
 **"7 21"**,  
 **"7 14"**};  
  
  
TreeFactory treeFactory = **new** TreeFactory();  
Tree<Integer> tree = treeFactory.createTreeFromStrings(input);

## 3.5. Други команди по дървото

### Добавяне на дете

Вземаме единият от алгоритмите за обхождане (в случая BFS), и в червен цвят е моята добавка

@Override  
**public void** addChild(E parentKey, Tree<E> child) {  
 Deque<Tree<E>> queue = **new** ArrayDeque<>();  
 **if** (**this**.**key** == parentKey) {  
 child.**parent** = **this**;  
 **this**.**children**.add(child);  
 **return**;  
 }  
  
 queue.offer(**this**);  
  
 **while** (queue.size() > 0) {  
 Tree<E> current = queue.poll();  
  
 **for** (Tree<E> ch : current.**children**) {  
 **if** (ch.**key** == parentKey) {  
 child.**parent** = ch;  
 ch.**children**.add(child);  
 **return**;  
 }  
 queue.offer(ch);  
 }  
 }  
}

**public static void** main(String[] args) {  
 Tree<Integer> tree =  
 **new** Tree<>(7,  
 **new** Tree<>(19,  
 **new** Tree<>(1),  
 **new** Tree<>(12),  
 **new** Tree<>(31)),  
 **new** Tree<>(21),  
 **new** Tree<>(14,  
 **new** Tree<>(23),  
 **new** Tree<Integer>(6))  
 );  
  
 tree.addChild(19, **new** Tree<Integer>(45));

}

### Връщане на поддърво

*//returns a subtree with root nodeKey***public** Tree<E> getNode(E nodeKey) {  
 **if** (**this**.**key**.equals(nodeKey)) {  
 **return this**;  
 }  
  
 Deque<Tree<E>> queue = **new** ArrayDeque<>();  
 queue.offer(**this**);  
  
 **while** (queue.size() > 0) {  
 Tree<E> current = queue.poll();  
  
 **for** (Tree<E> ch : current.**children**) {  
 **if** (ch.**key**.equals(nodeKey)) {  
 **return** ch;  
 }  
 queue.offer(ch);  
 }  
 }  
  
 **return null**;  
}

### Изтриване на поддърво

@Override  
**public void** removeNode(E nodeKey) {  
 **if** (**this**.**key** == nodeKey && **this**.**parent** == **null**) {  
 **this**.**children** = **new** ArrayList<>();  
 **this**.**key** = **null**;  
 **return**;  
 }  
  
 Tree<E> nodeToDelete = getNode(nodeKey);  
 **if** (nodeToDelete == **null**) {  
 **return**;  
 }  
  
 **if** (nodeToDelete.**children**.isEmpty()) {  
 List<Tree<E>> parentChildrenList = nodeToDelete.**parent**.**children**;  
 parentChildrenList.remove(nodeToDelete);  
 nodeToDelete.**parent** = **null**;  
 **return**;  
 } **else** {  
 List<Tree<E>> parentChildrenList = nodeToDelete.**parent**.**children**;  
 parentChildrenList.remove(nodeToDelete);  
 nodeToDelete.**parent** = **null**;  
 nodeToDelete.**children** = **new** ArrayList<>();  
 **return**;  
 }  
  
}

### Размени върхове

Направил съм го донякъде само

@Override  
**public void** swap(E firstKey, E secondKey) {  
 **if** (firstKey.equals(secondKey)) {  
 **return**;  
 }  
  
 Tree<E> firstSubtree = getNode(firstKey);  
 Tree<E> secondSubtree = getNode(secondKey);  
  
 **if** (firstSubtree.**parent** == **null**) {  
 **if** (secondSubtree.**children**.isEmpty()) { *//ако е листо* inTheParentListDeleteThatLeaf(secondSubtree);  
 secondSubtree.**children**.add(firstSubtree);  
 firstSubtree.**parent** = secondSubtree;  
 } **else** { *//ако е среден връх* }  
  
 **return**;  
 }  
  
 **if** (secondSubtree.**parent** == **null**) {  
 **if** (firstSubtree.**children**.isEmpty()) { *//ако е листо* inTheParentListDeleteThatLeaf(firstSubtree);  
 firstSubtree.**children**.add(secondSubtree);  
 secondSubtree.**parent** = firstSubtree;  
 }  
  
 **return**;  
 }  
  
 switchNodes(firstSubtree, secondSubtree);

}

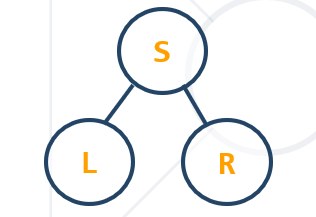
**private void** switchNodes(Tree<E> subtree1, Tree<E> subtree2) {  
 Tree<E> parent1 = subtree1.**parent**;  
 Tree<E> parent2 = subtree2.**parent**;  
 **int** indexSubtree1 = parent1.**children**.indexOf(subtree1);  
 **int** indexSubtree2 = parent2.**children**.indexOf(subtree2);  
  
 parent1.**children**.set(indexSubtree1, subtree2);  
 parent2.**children**.set(indexSubtree2, subtree1);  
 subtree1.**parent** = parent2;  
 subtree2.**parent** = parent1;  
}

**private void** inTheParentListDeleteThatLeaf(Tree<E> nodeToDelete) {  
 nodeToDelete.**parent**.**children**.remove(nodeToDelete); *//ако е листо* nodeToDelete.**parent** = **null**; *//ако е листо*}

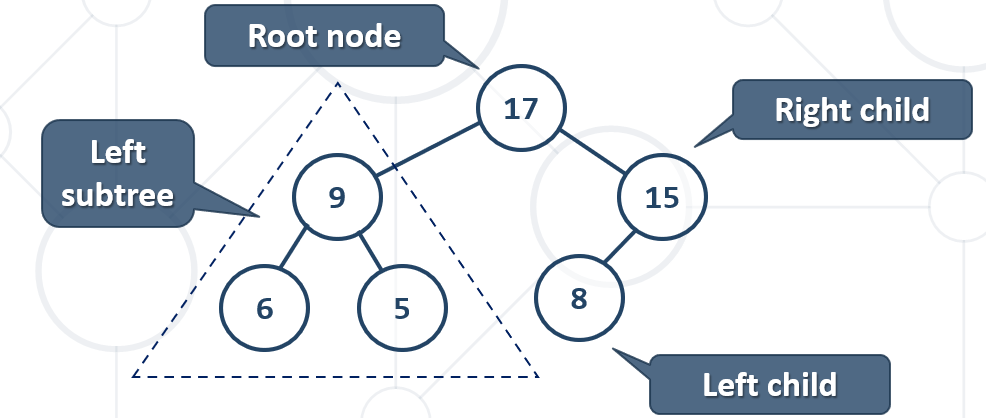
# 4. Binary Trees, Heaps and BST

## 4.1. Binary Tree – двоично дърво – неподредено / небалансирано !!!

* Each node has **at most two** children
  + Children are called **left** and **right**
  + The **parent** is also called **source**

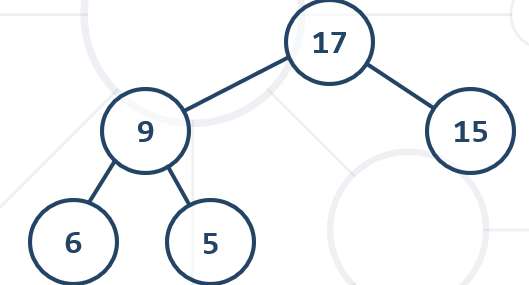


* **Binary trees**: the most widespread form - Each node **has at most 2 children** (**left** and **right**)

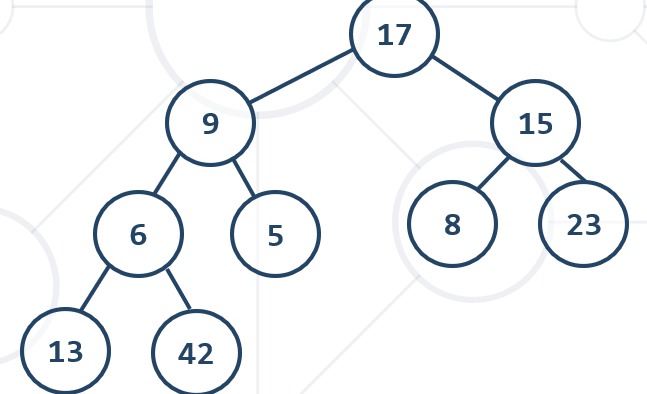


### Types of Binary Trees

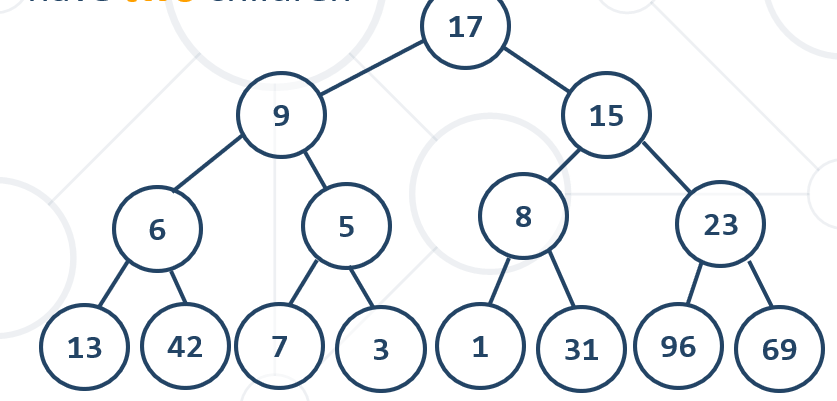
* **Full / Запълнено** – each node has **0** or **2** children - **всичко освен листата има деца**



* **Complete /** – nodes are filled **top** to **bottom** and **left** to **right – на всяко едно ниво е еднакво, без тавана/последното ниво**



* **Perfect** – combines **complete** and **full**
  + leafs are at the **same** **level**, other nodes have **two** children



### Инициализиране на двуично дърво

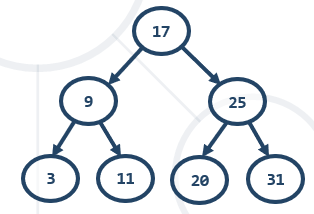
#### Вариант 1 – ръчно инициализиране / hardcore-нати стойности

**public class** BinaryTree<E> **implements** AbstractBinaryTree<E> {  
 **private** E **key**;  
 **private** BinaryTree<E> **left**;  
 **private** BinaryTree<E> **right**;  
  
 **public** BinaryTree(E key, BinaryTree<E> left, BinaryTree<E> right) {  
 **this**.**key** = key;  
 **this**.**left** = left;  
 **this**.**right** = right;  
 }

}

main

BinaryTree tree = **new** BinaryTree<>(17,  
 **new** BinaryTree<>(9, **new** BinaryTree<>(3, **null**, **null**),  
 **new** BinaryTree<>(11, **null**, **null**)),  
 **new** BinaryTree<>(25, **new** BinaryTree<>(20, **null**, **null**),  
 **new** BinaryTree<>(31, **null**, **null**))  
);



#### Вариант 2 – все пак ги подреждаме стойностите – отляво по-малки, отдясно по-големи

**public class** MessagingSystem **implements** DataTransferSystem {  
  
 **static class** Node {  
 Message **message**;  
 Node **left**;  
 Node **right**;  
  
 **public** Node(Message message) {  
 **this**.**message** = message;  
 }  
  
 **int** getWeight() {  
 **return this**.**message**.getWeight();  
 }  
 }  
  
 Node **root**;  
 **int size**;

@Override  
**public void** add(Message message) {  
 **if** (**root** == **null**) {  
 **root** = **new** Node(message);  
 } **else** {  
 add(**root**, message);  
 }  
  
 **size**++;  
}  
  
**private void** add(Node node, Message message) {  
 **if** (node.getWeight() == message.getWeight()) {  
 **throw new** IllegalArgumentException();  
 }  
  
 **if** (message.getWeight() < node.getWeight()) {  
 **if** (node.**left** == **null**) {  
 node.**left** = **new** Node(message);  
 } **else** {  
 add(node.**left**, message);  
 }  
 } **else** {  
 **if** (node.**right** == **null**) {  
 node.**right** = **new** Node(message);  
 } **else** {  
 add(node.**right**, message);  
 }  
 }  
}

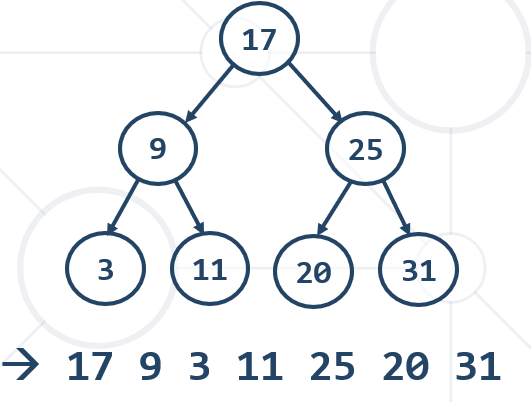
### Traversing binary tree

#### Pre-Order – по нормален начин – a kind of DFS

**Root -> Left -> Right**

First we **add** the **visiting** node then we **continue** with the **left** and **right** child

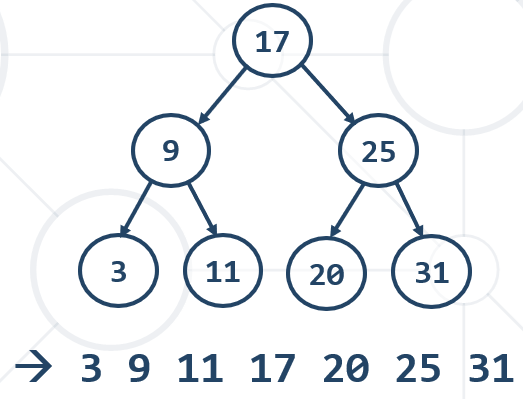
preOrder (node) {  
 **if** (node != **null**) {  
 print node.key  
 preOrder(node.left)  
 preOrder(node.right)  
 }  
}



#### In-Order - a kind of DFS

**Left 🡪 Root 🡪 Right**

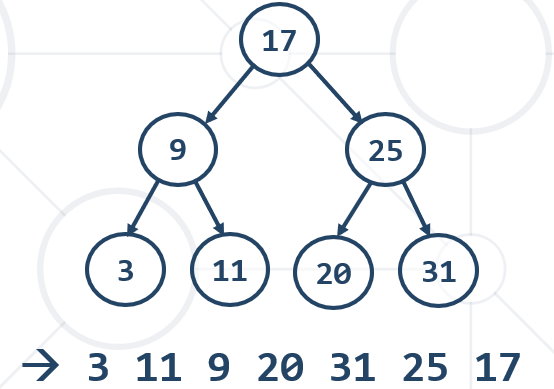
inOrder (node) {  
 **if** (node != **null**) {  
 inOrder(node.left)  
 print node.key  
 inOrder(node.right)  
 }  
}



#### Post-Order - a kind of DFS

**Left 🡪 Right 🡪 Root**

postOrder (node) {  
 **if** (node != **null**) {  
 postOrder(node.left)  
 postOrder(node.right)  
 print node.key  
 }  
}



### How to copy a tree

#### Pre-Order – по нормален начин

**public** BinaryTree<E> copyTree(){  
 **return** copy(**this**); *//това е корена*}  
  
**private** BinaryTree<E> copy(BinaryTree<E> root) {  
 **if** (root == **null**) {  
 **return null**;  
 }  
  
 BinaryTree<E> copiedTree = **new** BinaryTree<E>(root.getKey(), **null**, **null**);  
 copiedTree.**left** = copy((BinaryTree<E>) root.getLeft());  
 copiedTree.**right** = copy((BinaryTree<E>) root.getRight());

**return** copiedTree;  
}

main

BinaryTree tree = **new** BinaryTree<>(17,  
 **new** BinaryTree<>(9, **new** BinaryTree<>(3, **null**, **null**),  
 **new** BinaryTree<>(11, **null**, **null**)),  
 **new** BinaryTree<>(25, **new** BinaryTree<>(20, **null**, **null**),  
 **new** BinaryTree<>(31, **null**, **null**))  
);  
  
BinaryTree<Integer> secondCopiedTree = tree.copyTree();

### How to foreach a binary tree

The case of in-order

@Override  
**public void** forEachInOrder(Consumer<E> consumer) {  
 **if** (**this**.getLeft() != **null**) {  
 **this**.getLeft().forEachInOrder(consumer);  
 }  
 consumer.**accept**(**this**.getKey());  
  
 **if** (**this**.getRight() != **null**) {  
 **this**.getRight().forEachInOrder(consumer);  
 }  
}

StringBuilder builder = **new** StringBuilder();  
**tree**.forEachInOrder(key -> builder.append(key).append(**", "**));  
String actual = builder.toString();

### How to delete a node in a (binary) tree

#### Post-Order – изтриваме левия, десния, и се връщаме на бащата

Ако искаме да върнем всички изтрити елементи, то не можем просто да изтрием връзката, а трябва да изтрием всички под/sub-nodes. И тогава използваме Post-Order

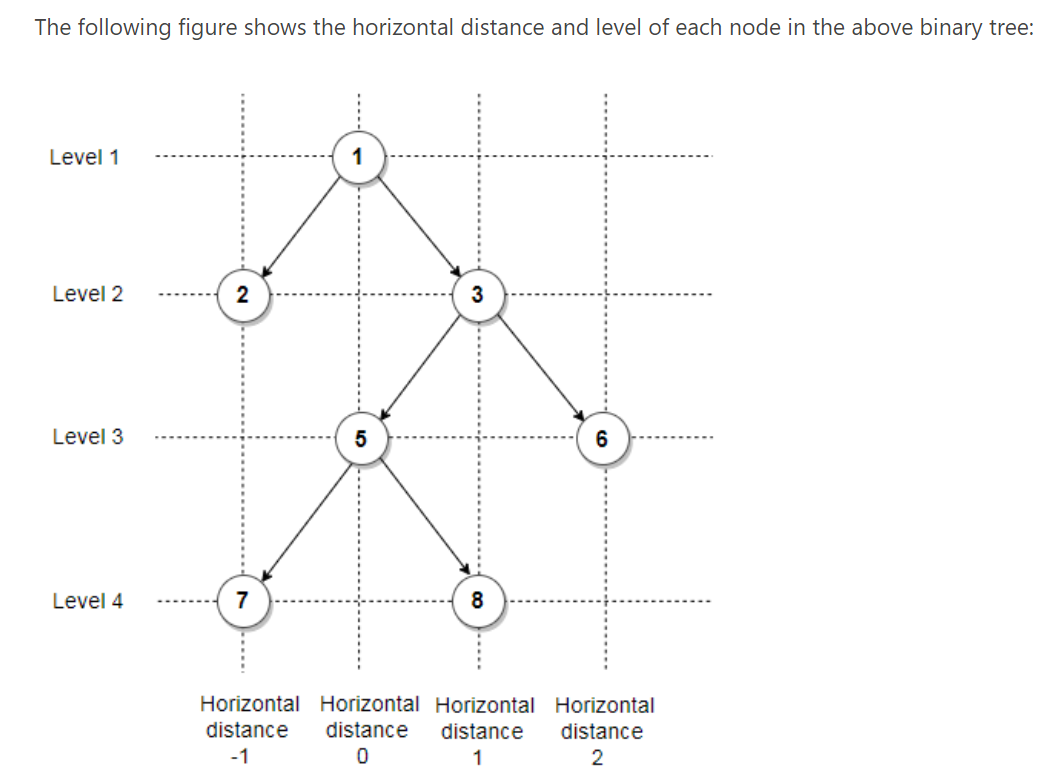
### Lowest Common Ancestor algorithm

In other words you can **ignore** the **value** you **should only care for the distance**.

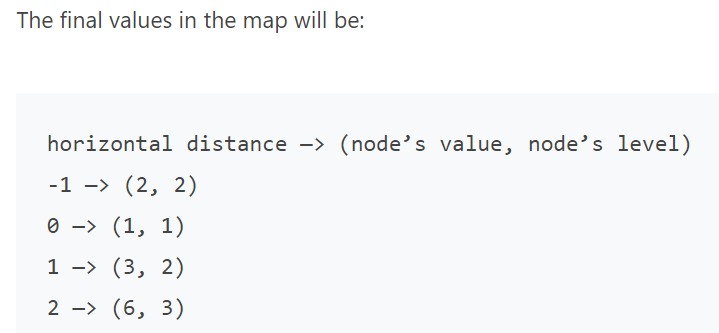
**public class** BinaryTree {  
 **private int value**;  
 **private** BinaryTree **left**;  
 **private** BinaryTree **right**;  
 **private** BinaryTree **parent**;  
  
  
 **public** BinaryTree(**int** key, BinaryTree left, BinaryTree right) {  
 **this**.**value** = key;  
 **this**.**left** = left;  
 **this**.**right** = right;  
 **this**.setParent(**null**);  
 **if** (**this**.**left** != **null**) {  
 **this**.**left**.setParent(**this**);  
 }  
 **if** (**this**.**right** != **null**) {  
 **this**.**right**.setParent(**this**);  
 }  
 }

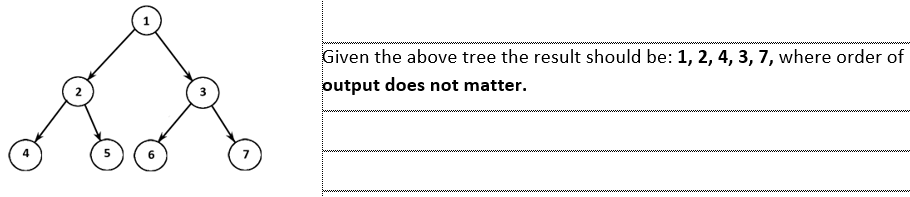
**private** BinaryTree findNode(BinaryTree current, **int** value) {  
 **if** (current == **null**) {  
 **return null**;  
 }  
  
 **if** (current.getValue() == value) {  
 **return** current;  
 } **else** {  
 BinaryTree foundNode = **this**.findNode(current.getLeft(), value);  
  
 **if** (foundNode == **null**) {  
 foundNode = **this**.findNode(current.getRight(), value);  
 }  
  
 **return** foundNode;  
 }  
}  
  
**private** List<BinaryTree> findAncestors(**int** value) {  
 List<BinaryTree> result = **new** ArrayList<>();  
 BinaryTree foundNode = **this**.findNode(**this**, value);  
  
 **while** (foundNode.getParent() != **null**) {  
 foundNode = foundNode.getParent();  
 result.add(foundNode);  
 }  
  
 **return** result;  
}  
  
**public** Integer findLowestCommonAncestor(**int** first, **int** second) {  
 List<BinaryTree> firstAncestors = **this**.findAncestors(first);  
 List<BinaryTree> secondAncestors = **this**.findAncestors(second);  
  
 **for** (**int** i = 0; i < firstAncestors.size(); i++) {  
 **if** (secondAncestors.contains(firstAncestors.get(i))) {  
 **return** firstAncestors.get(i).getValue();  
 }  
 }  
  
 **return null**;  
}

### **Top view elements algorithm**



**Horizontal distance = offset**





**public class** Pair<K, V> {  
 **private** K **key**;  
 **private** V **value**;  
  
 **public** Pair(K key, V value) {  
 **this**.**key** = key;  
 **this**.**value** = value;  
 }  
  
 **public** K getKey() {  
 **return key**;  
 }  
  
 **public void** setKey(K key) {  
 **this**.**key** = key;  
 }  
  
 **public** V getValue() {  
 **return value**;  
 }  
  
 **public void** setValue(V value) {  
 **this**.**value** = value;  
 }  
}

**public class** BinaryTree {  
 **private int value**;  
 **private** BinaryTree **left**;  
 **private** BinaryTree **right**;  
 **private** BinaryTree **parent**;

**public** BinaryTree(**int** key, BinaryTree left, BinaryTree right) { ……..}

**public** List<Integer> topView() {

**//първият елемент е offset – заема стойности от -3 до +3**

**//вторият параметър е стойността на върха**

**//третият елемент е нивото, на което се намира върха**   
 Map<Integer, Pair<Integer, Integer>> offsetToValueLevel = **new** TreeMap<>();  
  
 traverseTree(**this**, 0, 1, offsetToValueLevel);  
  
 List<Integer> collect = offsetToValueLevel.values()  
 .stream()  
 .map(e -> e.getKey())  
 .collect(Collectors.*toList*());  
  
 **return** collect;  
}  
  
**private void** traverseTree(BinaryTree tree, **int** offset, **int** level, Map<Integer,  
 Pair<Integer, Integer>> offsetToValueLevel) {  
 **if** (tree == **null**) {  
 **return**;  
 }  
  
 Pair<Integer, Integer> currentValueLevel = offsetToValueLevel.get(offset);  
 **if** (currentValueLevel == **null** || level < currentValueLevel.getValue()) {  
 offsetToValueLevel.put(offset, **new** Pair<>(tree.**value**, level));  
 }  
  
 traverseTree(tree.**left**, offset - 1, level + 1, offsetToValueLevel);  
 traverseTree(tree.**right**, offset + 1, level + 1, offsetToValueLevel);  
}

## 4.4. Binary Search Trees – подредено!!!

**Двоичните дървета за търсене предразполагат много добре за подреждане на елементи.**

**Приема се, че няма повтарящи се елементи!!!**

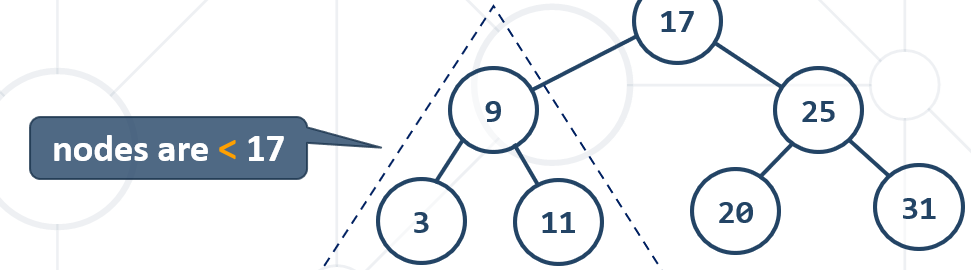
**Всички двуични дървета имат опeрaции с логаритмична сложност**

find -> O(log(n))

insert -> O(log(n))

delete -> O(log(n))

* **Binary search trees** are **ordered**
  + For each node ***x***
    - Elements in left subtree of ***x*** are ***< x***
    - Elements in right subtree of ***x*** are ***> x***



### BST – Insert - инициализация на BinarySearchTree

* + if node is **null** 🡪 insert x
  + else if x **<** node.value 🡪 **go left**
  + else if x **>** node.value 🡪 **go right**
  + else 🡪 node **exists**

#### Вариант с една стойност на Node

**public interface** AbstractBinarySearchTree<E **extends** Comparable<E>> {

**public static class** Node<E> {  
 **public** E **value**;  
 *// public E ;* **public** Node<E> **leftChild**;  
 **public** Node<E> **rightChild**;  
  
 **public** Node(E value) {  
 **this**.**value** = value;  
 }

}

**public class** BinarySearchTree<E **extends** Comparable<E>> **implements** AbstractBinarySearchTree<E> {  
 **private** Node<E> **root**;

**private** Node<E> **leftChild**;  
 **private** Node<E> **rightChild**;

**public static class** Node<E> {  
 **private** E **value**;  
 **private** Node<E> **leftChild**;  
 **private** Node<E> **rightChild**;  
  
 **public** Node() {  
 }  
  
 **public** Node(E value) {  
 **this**.**value** = value;  
 }  
  
 **public** Node(E value, Node<E> leftChild, Node<E> rightChild) {  
 **this**.**value** = value;  
 **this**.**leftChild** = leftChild;  
 **this**.**rightChild** = rightChild;  
 }  
  
 **public void** setLeftChild(Node<E> leftChild) {  
 **this**.**leftChild** = leftChild;  
 }  
  
 **public void** setRightChild(Node<E> rightChild) {  
 **this**.**rightChild** = rightChild;  
 }  
  
 **public** Node<E> getLeft() {  
 **return this**.**leftChild**;  
 }  
  
 **public** Node<E> getRight() {  
 **return this**.**rightChild**;  
 }  
  
 **public** E getValue() {  
 **return this**.**value**;  
 }  
}

**public** BinarySearchTree(E value) {  
**this**.**root** = **new** Node<E>(value);  
}

**public** BinarySearchTree() {  
}

}

@Override  
 **public void** insert(E key) {  
 Node<E> node = **new** Node<>(key, **null**, **null**);  
  
 **if** (**this**.getRoot() == **null**) {  
 **this**.**root** = node;  
 } **else** {  
 *//* ***TODO: Find the place to insert*** insertRecursive(key, **this**.**root**);  
 }  
 }

**private void** insertRecursive(E key, Node<E> node) {  
 **int** compareResult = key.compareTo(node.**value**);  
  
 **if** (compareResult == 0) {  
 **return**;  
 }  
  
 **if** (compareResult < 0) {  
 **if** (node.**leftChild** == **null**) {  
 node.**leftChild** = **new** Node<>(key, **null**, **null**);  
 } **else** {  
 insertRecursive(key, node.**leftChild**);  
 }  
  
 } **else** {  
 **if** (node.**rightChild** == **null**) {  
 node.**rightChild** = **new** Node<>(key, **null**, **null**);  
 } **else** {  
 insertRecursive(key, node.**rightChild**);  
 }  
 }  
}

}

main

BinarySearchTree<Integer> bst = **new** BinarySearchTree<>();  
bst.insert(12);  
bst.insert(21);  
bst.insert(5);  
bst.insert(1);  
bst.insert(8);  
bst.insert(18);  
bst.insert(23);

#### Вариант с две стойности на Node = TreeMap

**public interface** AbstractBinarySearchTree<E **extends** Comparable<E>> {

**public static class** Node<E> {  
 **public** E **key**;  
 **public** E **value**;**public** Node<E> **leftChild**;  
 **public** Node<E> **rightChild**;  
  
 **public** Node(E key, E value) {  
 **this**.**key** = key;

**this**.**value** = value;  
 }  
  
 **public** Node(E key, E value, Node<E> leftChild, Node<E> rightChild) {  
 **this**.**key** = key;

**this**.**value** = value;  
 **this**.**leftChild** = leftChild;  
 **this**.**rightChild** = rightChild;  
 }

}

**public class** BinarySearchTree<E **extends** Comparable<E>> **implements** AbstractBinarySearchTree<E> {  
 **private** Node<E> **root**;  
  
 @Override  
 **public void** insert(E key, E value) {  
 Node<E> node = **new** Node<>(key, value, **null**, **null**);  
  
 **if** (**this**.getRoot() == **null**) {  
 **this**.**root** = node;  
 } **else** {  
 *//* ***TODO: Find the place to insert*** insertRecursive(key, value, **this**.**root**);  
 }  
 }

**private void** insertRecursive(E key, E value, Node<E> node) {  
 **int** compareResult = key.compareTo(node.**key**); //сравняваме по кеу  
  
 **if** (compareResult == 0) {  
 **return**;  
 }  
  
 **if** (compareResult < 0) {  
 **if** (node.**leftChild** == **null**) {  
 node.**leftChild** = **new** Node<>(key, value, **null**, **null**);  
 } **else** {  
 insertRecursive(key, value, node.**leftChild**);  
 }  
  
 } **else** {  
 **if** (node.**rightChild** == **null**) {  
 node.**rightChild** = **new** Node<>(key, value, **null**, **null**);  
 } **else** {  
 insertRecursive(key, value, node.**rightChild**);  
 }  
 }  
}

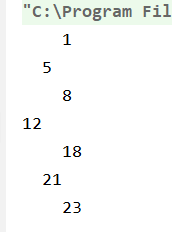
}

#### In-order (**Left 🡪 Root 🡪 Right**) на двоично дърво за търсене връща елементите в сортиран вид!!!

main

BinarySearchTree<Integer> bst = **new** BinarySearchTree<>();  
bst.insert(12);  
bst.insert(21);  
bst.insert(5);  
bst.insert(1);  
bst.insert(8);  
bst.insert(18);  
bst.insert(23);

**bst.print(); връща**



**public void** print(){  
 printRecursive(**this**.**root**, 0);  
 }  
  
 **private void** printRecursive(Node<E> node, **int** level){  
 **if** (node == **null**) {  
 **return**;  
 }  
  
 StringBuilder padding = **new** StringBuilder();  
 **for** (**int** i = 0; i < level; i++) {  
 padding.append(**" "**);  
 }  
  
 printRecursive(node.**leftChild**, level + 1);  
 System.***out***.println(padding.append(node.getValue()));  
 printRecursive(node.**rightChild**, level + 1);  
 }  
}

### How to foreach a binary search tree(BST)

The case of in-order

**public void** eachInOrder(Consumer<E> consumer) {  
 **this**.internalEachInOrder(**this**.**root**, consumer);  
}  
  
**private void** internalEachInOrder(Node<E> node, Consumer<E> consumer) {  
 **if** (node == **null**) {  
 **return**;  
 }  
  
 **this**.internalEachInOrder(node.getLeft(), consumer);  
 consumer.accept(node.getValue());  
 **this**.internalEachInOrder(node.getRight(), consumer);  
}

### BST - Search

* Search for **x** in BST
  + if node is not null
    - if x **<** node.value 🡪 **go left**
    - else if x **>** node.value 🡪 **go right**
    - else if x **==** node.value 🡪 **return**

#### Вариант 1 – смесен вариант

@Override  
**public** AbstractBinarySearchTree<E> search(E searchedElement) {  
 AbstractBinarySearchTree<E> result = **new** BinarySearchTree<>();  
 Node<E> current = **this**.**root**;  
 **while** (current != **null**){  
 **if**(searchedElement.compareTo(current.**value**) < 0){  
 current = current.**leftChild**;  
 } **else if**(searchedElement.compareTo(current.**value**) > 0){  
 current = current.**rightChild**;  
 } **else** { *// ако елемента съвпада* **return new** BinarySearchTree<E>(current);  
 }  
 }  
 **return** result;  
}  
  
*//генерирай дърво с метода insert с определен корен***private** BinarySearchTree(Node<E> root) {  
 **this**.copy(root);  
}  
  
*//генерирай дърво с метода insert с определен корен – ИЗПОЛЗВА Pre-Order***private void** copy(Node<E> node) {  
 **if**(node!=**null**) {  
 **this**.insert(node.**value**);  
 **this**.copy(node.**leftChild**);  
 **this**.copy(node.**rightChild**);  
 }  
}

#### 2ри вариант само с рекурсия за търсене на елемент

**private** Node<E> internalSearchReturnNodeRecursive(Node<E> node, E element) {  
 **if** (node == **null**) {  
 **return null**;  
 }  
 **if** (node.getValue().compareTo(element) < 0) {  
 **return this**.internalSearchReturnNodeRecursive(node.getRight(), element);  
 } **else if** (node.getValue().compareTo(element) > 0) {  
 **return this**.internalSearchReturnNodeRecursive(node.getLeft(), element);  
 }  
   
 **return** node;  
}

**public** BinarySearchTree<E> search(E searchedElement) {

Node<E> current = internalSearchReturnNodeRecursive(this.root, element);

return current == null ? null :

new BinarySearchTree<E>(current.getValue());

}

### BST – Contains

**Итеративен вариант, съществува и рекурсивен вариант**

@Override  
**public boolean** contains(E element) {  
 Node<E> current = **this**.**root**;  
 **while** (current != **null**){  
 **if** (element.compareTo(current.**value**) < 0){  
 current = current.**leftChild**;  
 } **else if** (element.compareTo(current.**value**) > 0){  
 current = current.**rightChild**;  
 } **else** {  
 **break**;  
 }  
 }  
 **return** current != **null**;  
}

* Binary search trees can be **balanced**
  + Balanced trees have for each node
    - Nearly equal number of nodes in its subtrees

**Balanced trees** have **height of ~ log(n)**

### BST – deleteMin

**public void** deleteMin() {  
 isEmptyTree();  
  
 **if** (**this**.**root**.getLeft() == **null**) {  
 **this**.**root** = **this**.**root**.getRight();  
 } **else** {  
 Node<E> current = **this**.**root**;  
  
 **while** (current.getLeft().getLeft() != **null**){  
 current = current.getLeft();  
 }  
  
 current.setLeftChild(current.getLeft().getRight());  
 }  
}

### BST – deleteMax

**public void** deleteMax() {  
 isEmptyTree();  
  
 **if** (**this**.**root**.getRight() == **null**) {  
 **this**.**root** = **this**.**root**.getLeft();  
 } **else** {  
 Node<E> current = **this**.**root**;  
  
 **while** (current.getRight().getRight() != **null**){  
 current = current.getRight();  
 }  
  
 current.setRightChild(current.getRight().getLeft());  
 }  
}

### BST – count elements

**public int** count() {  
 **return this**.internalCount(**this**.**root**);  
}  
  
**private int** internalCount(Node<E> node) {  
 **if** (node == **null**) {  
 **return** 0;  
 } **else** {  
 **return** 1 + (node.getLeft() == **null** ? 0 : **this**.internalCount(node.getLeft()))  
 + (node.getRight() == **null** ? 0 : **this**.internalCount(node.getRight()));  
 }  
}

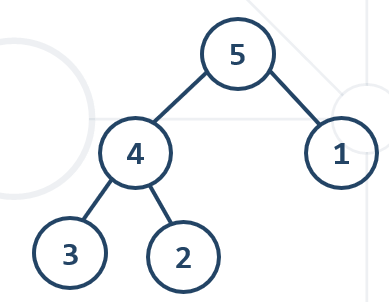
## 4.2. Heap – пирамида/куп

### What is Heap?

* **Heap**
  + Tree-based data structure
  + Stored in an array
* Heaps hold the **heap property** for each node:
  + **Min Heap**
    - parent ≤ children
  + **Max Heap**
    - parent ≥ children

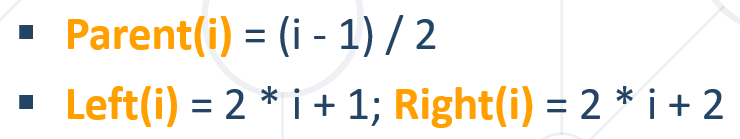
Няма изискване отляво да са по-малки, а отдясно да са по-големи

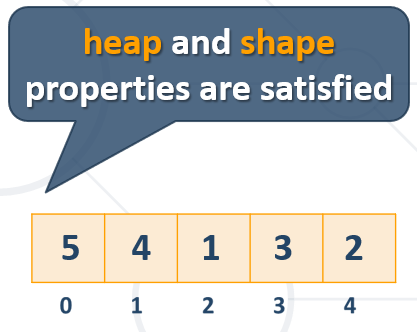
* **Binary heap**
  + Represents a Binary Tree
* **Shape property** - Binary heap is a **complete binary tree**:
  + - Every level, except the last, is **completely filled**
    - Last is filled **from left to right**



### Binary Heap – Array Implementation

* Binary heap can be efficiently stored in an array

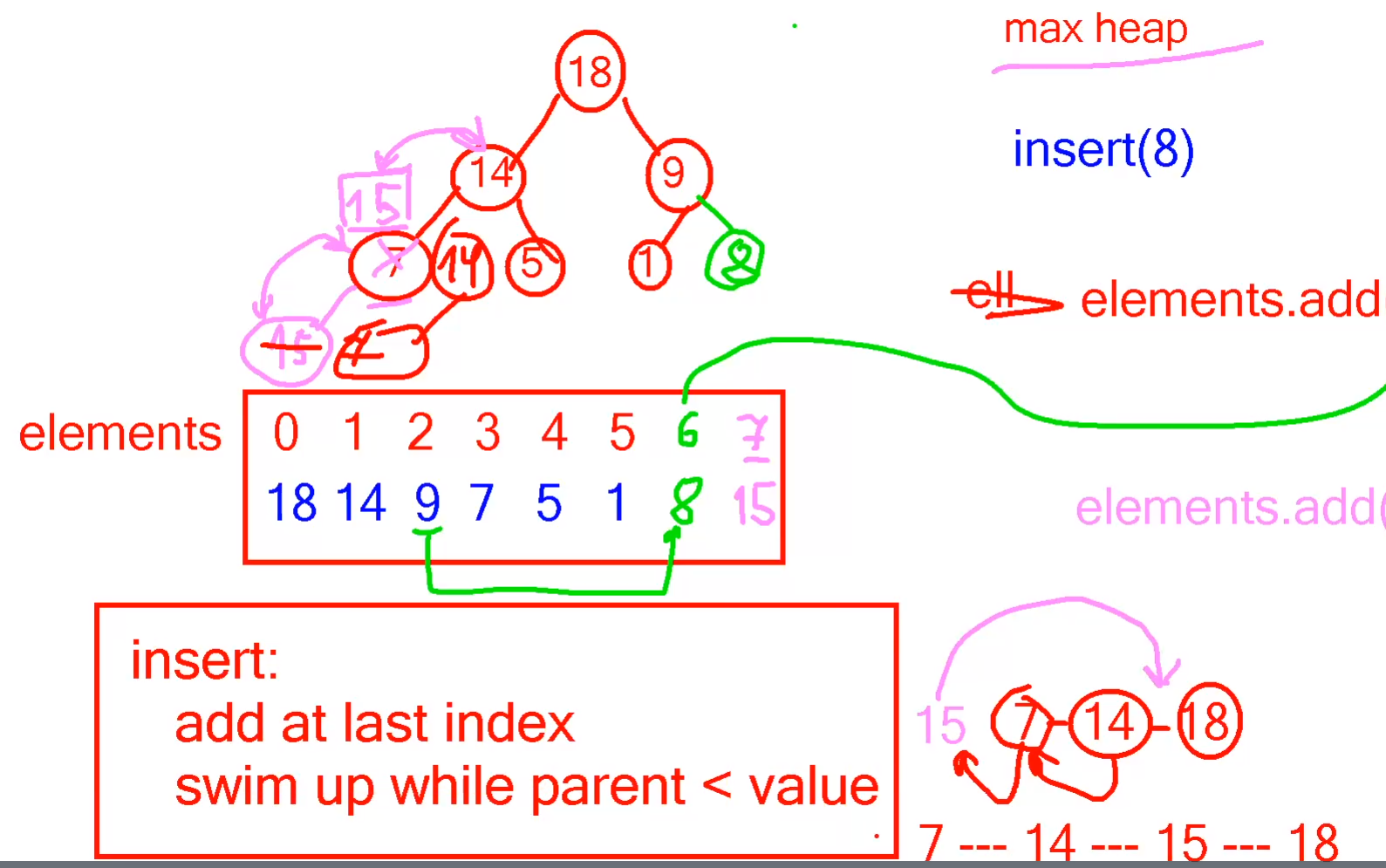




#### Heap Insertion = Initialization на МaxHeap

Insert element -> add last and SWIM

* To preserve **heap properties**:
  + **Insert** at the end
  + **Heapify** element up



**public class** MaxHeap<E **extends** Comparable<E>> **implements** Heap<E> {  
 **private** List<E> **elements**;  
  
 **public** MaxHeap() {  
 **this**.**elements** = **new** ArrayList<>();  
 }

@Override  
**public void** add(E element) {  
 **this**.**elements**.add(element);  
 **this**.heapifyUp(**this**.size() - 1); //heapifyUp = SWIM = изплува нагоре ако има нужда  
}

**private void** heapifyUp(**int** index) {  
 **while** (hasParent(index) && less(parent(index), **elements**.get(index))) {  
 **int** parentAt = getParentAt(index);  
 Collections.*swap*(**this**.**elements**, parentAt, index);  
 index = parentAt;  
 }  
}

**private int** getParentAt(**int** index) {  
 **return** (index - 1) / 2;  
}  
  
**private boolean** less(E parent, E child) {  
 **int** result = parent.compareTo(child); //ако parent е по-малък от детето, което изплува  
 **if** (result < 0) {  
 **return true**;  
 }  
  
 **return false**;  
}  
  
**private** E parent(**int** index) {  
 **return this**.**elements**.get((index - 1) / 2);  
}  
  
**private boolean** hasParent(**int** index) {  
 **if** (index == 0) {  
 **return false**;  
 }  
  
 **return true**;  
}

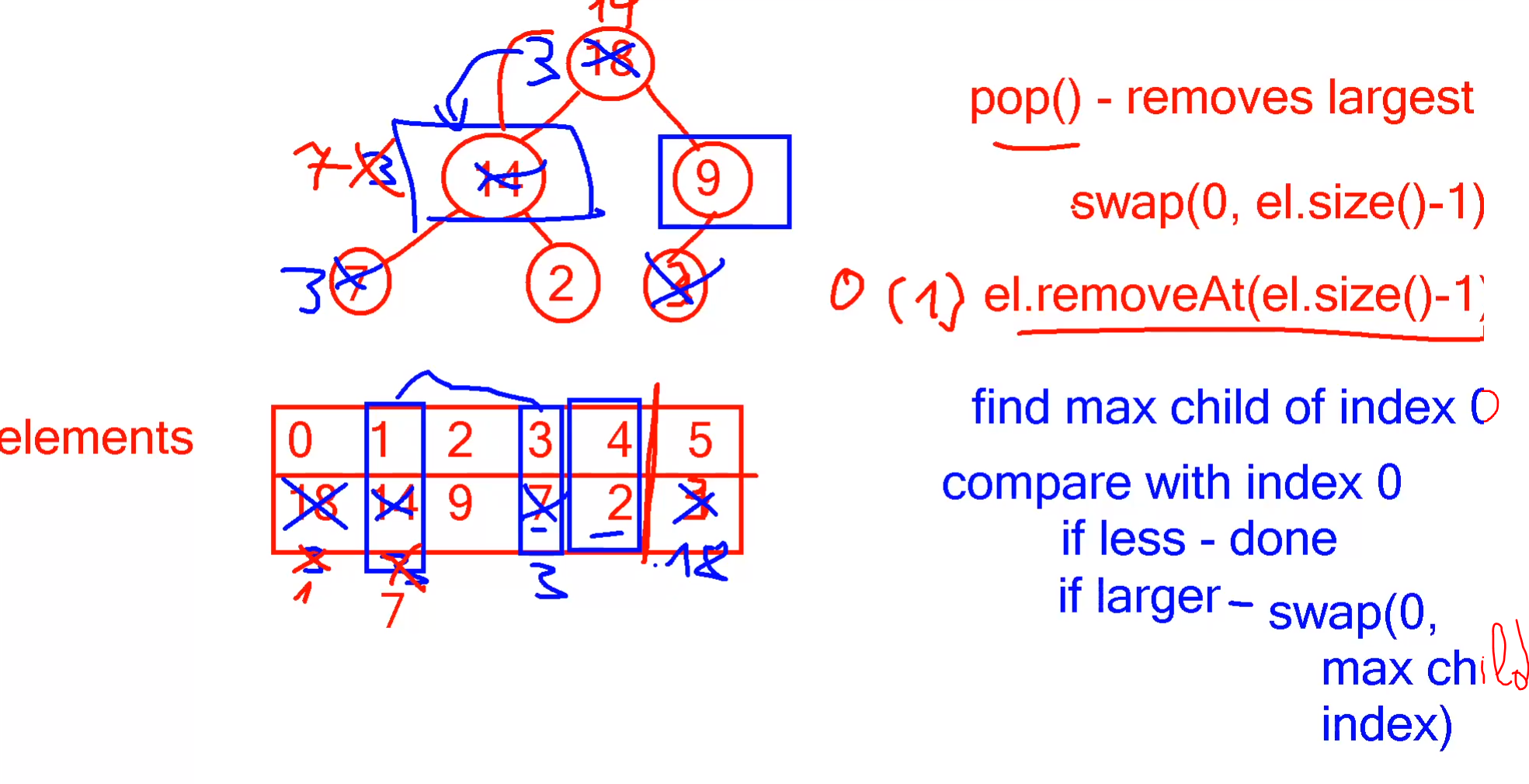
#### Heap pеек – get the 1st element in the array – МaxHeap and MinHeap

@Override  
**public** E peek() {  
 **if** (**this**.size() == 0) {  
 **throw new** IllegalStateException(**"Heap is empty upon peek attempt"**);  
 }  
 **return this**.**elements**.get(0);  
}

#### Heap pop/poll – removes largest element - МaxHeap

**Кофти се имплементира с код – да видя имплентацията на PriorityQueue**

Pop max element -> swap first and last; resize size-1 (намаляваме масива с 1 – без последния елемент); SINK first



#### Heap decrease element – the MinHeap case

@Override  
**public void** decrease(E element) {  
 **int** decreasedElementIndex = **this**.**elements**.indexOf(element);  
 E heapElement = **this**.**elements**.get(decreasedElementIndex);  
 heapElement.decrease(); //it decreases by 1  
  
 **this**.heapifyUp(decreasedElementIndex);  
}

Или

@Override  
**public void** decrease(E element) {

**this**.elements.remove(element);  
 element.decrease(); //it decreases by 1  
  
 **this**.add(element);  
}

## 4.3. Priority Queue – най-често се реализира като **Heap!!!**

**При приоритетната опашка, най-големият елемент/приоритетен отива най-първи на върха**

### Dequeue Most Significant Element

* ADS representing queue or stack like DS
  + Each element is **served** in **priority**
  + High priority is served **before** low priority
  + Elements with **equal** priority
  + Served in **order** of **input** or **undefined**
* Retains a **specific order** to the elements
* **Higher priority** elements are **pushed to the beginning** of the queue
* **Lower priority** elements are **pushed to the end** of the queue
* **Priority queue** abstract data type (ADT) supports:
  + **Insert(element)**
  + **Pull()** 🡪 **max**/**min** **element**
  + **Peek()** 🡪 **max**/**min** **element**
* Where **element** has a priority

**Priority**

* In Java usually the priority is passed as comparator
  + E.g. **Comparable<E>**

### PriorityQueue – Array Implementation

#### Insertation/initialization – like the MaxHeap

Insert element -> add last and SWIM

**public class** PriorityQueue<E **extends** Comparable<E>> **implements** AbstractQueue<E> {  
 **private** List<E> **elements**;  
  
 **public** PriorityQueue() {  
 **this**.**elements** = **new** ArrayList<>();  
 }

etc.

#### PriorityQueue peek – like the Heap

@Override  
**public** E peek() {  
 ensureNonEmpty();  
 **return this**.**elements**.get(0);  
}  
  
**private void** ensureNonEmpty() {  
 **if** (**this**.size() == 0) {  
 **throw new** IllegalStateException(**"Heap is empty upon peek/poll attempt"**);  
 }  
}

#### PriorityQueue poll – removes largest element - МaxHeap

Pop/poll max element ->

swap first and last;

resize size-1 (намаляваме масива с 1 – без последния елемент);

SINK first

Save the element on the top of the heap (index 0), **swap** the **first** and **last elements**, **exclude** the **last element** and **demote** the one **at the top until it has correct position**

@Override  
**public** E poll() {  
 ensureNonEmpty();  
 E removedElement = **this**.**elements**.get(0);  
 Collections.*swap*(**this**.**elements**, 0, **this**.size() - 1);  
 **this**.**elements**.remove(**this**.**elements**.size() - 1);  
 **this**.heapifyDown(0);  
  
 **return** removedElement;  
}

*/\*function will demote the element at a given index until it has no children  
 or it is greater than its both children. The first check will be our loop condition\*/***private void** heapifyDown(**int** index) {  
 **while** (index < **this**.**elements**.size() / 2) {  
 **int** childLeftOrRightIndex = 2 \* index + 1;  
 **int** childRightIndex = 2 \* index + 2;  
  
 *//ако левия клон е по-малък от десния, до десния по-голям изплува нагоре* **if** (childRightIndex < **this**.**elements**.size() &&  
 less(**this**.**elements**.get(childLeftOrRightIndex), **this**.**elements**.get(childRightIndex))) {  
 childLeftOrRightIndex += 1;  
 }  
  
 *//ако детето стане първия елемент с нулев индекс* **if** (less(**this**.**elements**.get(childLeftOrRightIndex), **this**.**elements**.get(index))) {  
 **break**;  
 }  
  
 Collections.*swap*(**this**.**elements**, index, childLeftOrRightIndex);  
 index = childLeftOrRightIndex;  
 }  
}

## 4.5. О notation table comparison

