

Java lists, maps and not only...

Paweł Jaworski

Luxoft/Akademia Górniczo Hutnicza

2020-11-10

Outline

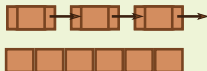
- 1 Hello
- 2 Algorithmics
 - Complexity calculation and nomenclature
- 3 List
 - ArrayList
 - Generics
 - LinkedList
- 4 Maps
 - HashMap
- 5 Exercices
 - Test Driven Development (TDD)
 - Laboratory exercices descriptions
- 6 Summary
 - What have we learned today?

Introduction

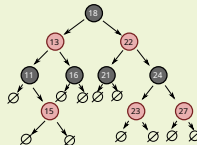
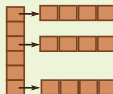
- Java collections are sophisticated implementations of algorithms for storing data (see also Cormen's "Introduction to algorithms")
- All collections implement iteration
- Lists are implemented as sequential collections backed by arrays and linked lists
- Sets, besides iteration, are also allowing checking if the item is present.
- Maps are associative containers which allow quick access by object-key (any object, also other than integer).
- Trees are underlying some implementations of Maps and Sets
- Lists maintain item order while it is not guaranteed in Sets and Maps.

The plan

Lists



Maps



$$\sum_{n=1}^{\infty} 0 < \left| \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} \right| < \infty$$

Algorithm analysis

The Big O, Ω , Θ notation

When there exists $n_0 > 0$ and $c > 0$ for which we can say that:

$$\forall n > n_0 \quad f(n) \leq c \cdot g(n)$$

we say that f is big-O g and write $f(x) = O(g(x))$

When there exists $n_0 > 0$ and $c > 0$ for which we can say that:

$$\forall n > n_0 \quad f(n) \geq c \cdot g(n)$$

we say that f is big- Ω g and write $f(x) = \Omega(g(x))$

The Big Θ notation

When we can say that $f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$ we say that f is Big Θ g and write:

$$f(x) = \Theta(g(x))$$

Examples

- Given $g(x) = x^2$ and $f(x) = 3x^2 + 4$ we can say that $f(x) = O(g(x)) \dots$
- Given $g(x) = x^3$ and $f(x) = 100x^2 + 4x^3$ we can say that $f(x) = O(g(x)) \dots$
- Given $g(x) = 2^x$ and $f(x) = 2^x + x^{64}$ we can say that $f(x) = O(g(x)) \dots$

We can also easily say that $x^2 = O(2^x)$, and $x^2 = \Omega(1)$ but it is not very interesting. This observation is very weak. We are rather interested in finding the closest matches.

Θ sufficient condition

The most interesting is finding the simple in form, but exact, match of the function. This match is symbolised by Θ .

$$f(x) = O(g(x)) \wedge f(x) = o(g(x)) \implies f(x) = \Theta(g(x))$$

If $f(x) = \Theta(g(x))$, it means that f grows asymptotically as fast as g .

Sufficient condition for $f(x) = \Theta(g(x))$ is:

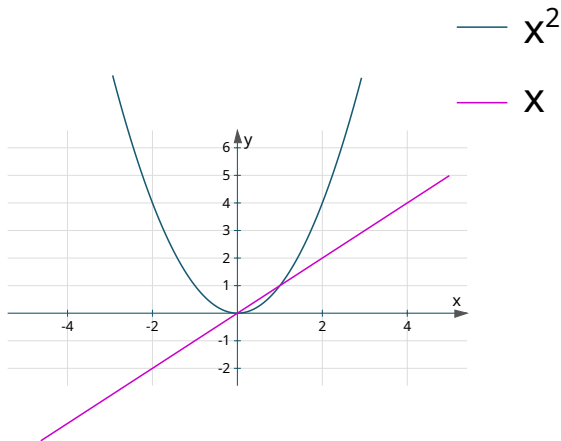
$$0 < \lim_{x \rightarrow \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$$

Some things that programmers usually know

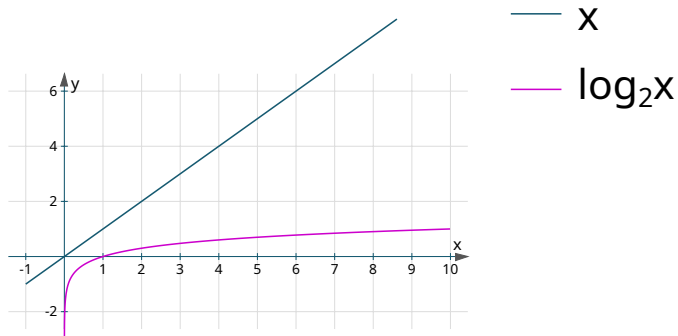
We use given notation extensively to describe algorithm time of execution (and memory consumption). Hence for us (generally):

- x^2 is worse than x
- x is worse than $\log x$
- 2^x is worse than x^{16} and it's bad
- $x!$ – it's so much, we don't distinguish between $x!$ and 2^x . Those are equally BAD.
- We often don't distinguish between logarithm and exponential bases

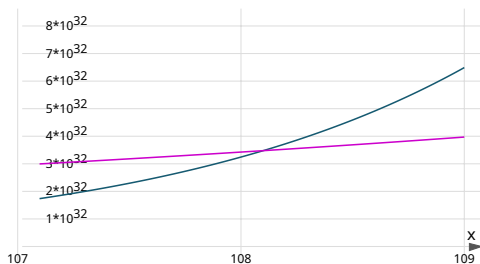
Examples



Examples



Examples



— 2^x
— x^{16}

Worst case scenario

Programmers also consider worst case (pessimistic) scenarios of algorithms and their complexity. Some algorithms work very slow on some special cases of input data. E.g. **quicksort**, despite of being $O(n \cdot \log n)$ runs at n^2 time, when in every partitioning selected pivot divides data to lengths: 1 and rest.

Having that in mind, we can select, for example, **heapsort**, which has worst case running time still $n \cdot \log n$

Need to know by heart

- *All arrays and lists in Java are indexed from 0!!!*

ArrayList

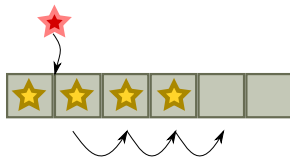
- Allocated as a one block in memory (more “array-ish” than “list-ish”)
- Quick “please get me an element at position n ” (further referred to as **get**): $\Theta(1)$
- Slow “please insert element at position n , moving all following elements to the right” (further referred to as **insert**): $O(n)$
- Slow “please delete element at position n , moving all following elements to the left” (further referred to as **delete**): $O(n)$

ArrayList

- Quick **append** - a special-case of the insert at the last place, i.e. when for n equal number of elements: $\Theta(1)$, but only when the underlying array size is $N > n + 1$
- When ArrayList is full, to perform insert we need to expand underlying array. We do it by increasing size by twice the current size.
- Is really the **append** operation $\Theta(1)$???

ArrayList pros and cons

- Insert at an arbitrary place costs n operations for $n = s - i$ where s is number of items and i is the position we insert at.



Amortised cost. Amortised analysis

- We analyse amortised cost of operations amortised by their number.
- Amortised cost for given $f(n)$ is $F(n)$ that:

$$F(n) = \frac{T(n)}{n}$$

where

$$T(n) = \sum_{x=0}^n f(x)$$

Amortised cost. Amortised analysis

- For array initialised to 2, in the **append** operation we have cost:

$$1, 1, n_1 = 2, 1, 1, n_2 = 4, 1, 1, 1, 1, n_3 = 8$$

where n_x is n-th resize cost equals array's size at the moment of resize.

- If we use an accounting method to tell that every operation, we put additional 2 operations' time on a special account, for a later use, we can reuse that time at critical sections of resizing. Our account state is then:

Amortised cost. Amortised analysis

Operation	Account state
+2	2
+2	4
-2	2
+2	4
+2	6
-4	2
+2	4
+2	6
+2	8
+2	10
-8	2

Table 1: Amortised analysis using accounting method

Amortised cost. Amortised analysis

SVG Animation for the amortised cost accounting method

Amortised cost. Amortised analysis

- In this case we say that operation **append** on ArrayList is $O(1)$ and keep in mind that sometimes it can stop our system for a very long time. So if we want to get overall computation time fit, we can allow us to expand a very large ArrayList, but when we are low-latency needers, we might need to search for a better solution.

Generics and wildcards

```
public void addToList(List<? extends Number> lst) {  
    Number item1 = lst.get(0);  
    lst.add(3);  
}
```

```
List<Float> floatList = ...;  
addToList(floatList);
```

Generics and wildcards

```
public void addToList(List<? extends Number> lst) {  
    Number item1 = lst.get(0); // ok  
    lst.add(3); // error  
}
```

```
List<Float> floatList = ...;  
addToList(floatList);
```

Generics and wildcards

```
public void addToList(List<? super Integer>) {  
    Integer i = lst.get(0);  
    lst.add(4);  
}
```

```
List<Number> numberList = ...;  
addToList(numberList);
```

Generics and wildcards

```
public void addToList(List<? super Integer>) {  
    Integer i = lst.get(0); // error  
    lst.add(4); // ok  
}
```

```
List<Number> numberList = ...;  
addToList(numberList);
```

Generics and wildcards

```
// https://stackoverflow.com/questions/4343202

public class Collections {
    public static <T> void copy(List<? super T> dest,
List<? extends T> src) {
        for (int i = 0; i < src.size(); i++)
            dest.set(i, src.get(i));
    }
}
```

Type erasure

```

1 public class Node<T> {
2     private T data;
3     private Node<T> next;
4
5     public Node(T data, Node<T> next) {
6         this.data = data;
7         this.next = next;
8     }
9
10    public void setData(T data) {
11        this.data = data;
12    }
13
14    public T getData() {
15        return data;
16    }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Node<Integer> node = new Node<>(10, null);
22         Integer i = node.getData();
23         node.setData(20);
24     }
25 }
26

```

```

1 public class Node {
2     private Object data;
3     private Node next;
4
5     public Node(Object data, Node next) {
6         this.data = data;
7         this.next = next;
8     }
9
10    public void setData(Object data) {
11        this.data = data;
12    }
13
14    public Object getData() {
15        return this.data;
16    }
17 }
18
19 public class Main {
20     public Main() {
21     }
22
23     public static void main(String[] args) {
24         Node node = new Node(10, (Node) null);
25         Integer i = (Integer) node.getData();
26         node.setData(20);
27     }
28 }
29

```

Figure 5: Type erasure

Type erasure

- Replace unbound type E with Object
- Add typecasting to all method calls
- Object has no information of type it was created with¹
- `ClassCastException` will be called if constraints are violated

¹except for class that inherits type-specified generic

LinkedList

- Allocated as linked list of many objects
- Every inserted object needs a wrapper object, so the number of objects in memory are at least **twice** the number of elements in list.
- Fast **append**: $\Theta(1)$
- Fast **insert** but only when given a **preceeding node** in the list.
- Fast **delete** but under the same conditions as **insert**
- Slow **get**: $O(n)$
- Does not need to grow - **append** is not lagging from time to time.

LinkedList - pros

- LinkedList implements Deque which gives us nice stack and fifo methods: pollFirst, pollLast, peekFirst, peekLast

LinkedList - cons

- Every item in LinkedList is of a class LinkedList.Item which has much overhead.

mark hash gcAge tLock tId	klassPtr ref to class	length length ptr	padding
4-8 bytes	4-8 bytes	4 bytes	up to mod arch

Remove all strings containing letter a

```
List<String> lst = new ArrayList<>();  
lst.addAll(Arrays.asList("marek", "basia", "kotek", "paw"  
    ));  
// remove all words containing letter a
```

Remove all strings containing letter a

```
List<String> lst = new ArrayList<>();
lst.addAll(Arrays.asList("marek", "basia", "kotek", "paw"
    ));
// remove all words containing letter a
for (String l : lst) {
    if (l.contains("a")) {
        lst.remove(l);
    }
}
```

Remove all strings containing letter a

```
List<String> lst = new ArrayList<>();
lst.addAll(Arrays.asList("marek", "basia", "kotek", "paw"
    ));
// remove all words containing letter a
for (String l : lst) { //
    ConcurrentModificationException
    if (l.contains("a")) {
        lst.remove(l);
    }
}
```

Hello
○○

Algorithmics
○○○○○○○○

List
○○○○○○○○○○○○○○○○○○○○●

Maps
○○○○○

Exercices
○○○○○○○

Summary
○○○○○

[LinkedList](#)

ListIterator

How much do the object weight in Java

```
class A {}

class B extends A {}

class C {
    boolean a;
}
```

How much do the object weight in Java - compressed pointers

```
java -XX:+UseCompressedOops
```

```
sizeof A = 16
```

```
sizeof B = 16
```

```
sizeof C = 16
```

```
sizeof Boolean = 16
```

```
sizeof Char = 16
```

```
sizeof Integer = 16
```

```
sizeof Long = 24
```

```
sizeof class java.util.LinkedList\$$Node = 24
```

How much do the object weight in Java - *not* compressed pointers

java -XX:-UseCompressedOops

sizeof A = 16

sizeof B = 16

sizeof C = 24

sizeof Boolean = 24

sizeof Char = 24

sizeof Integer = 24

sizeof Long = 24

sizeof class java.util.LinkedList\<Node> = 40

How much do the object weight in Java

```
public class C1 {  
    boolean a;  
    int b;  
} // size 24
```

```
public class C2 {  
    boolean a;  
    int b;  
    int c;  
} // size 24
```

```
public class C3 {  
    boolean a;  
    int b;  
    int c;  
} // size 32
```

HashMap

- Fast finding element by the key (further referred to as: **lookup**): $O(1)$.²
- Fast putting a key-value pair (further referred to as: **insert**): ($O(1)$, pessimistically $n = \text{size}$, when map needs to grow).
- Fast remove: $O(1)$ (HashMap does not shrink).
- Values with the same hash stored in *LinkedList*, collisions may occur.

²When time is longer, in properly setup map (i.e. number of buckets $>$ size), it means that collisions occur. This can be the sign of incorrect hash function.

HashMap

$x.\text{equals}(y) \implies x.\text{hashCode}() == y.\text{hashCode}()$

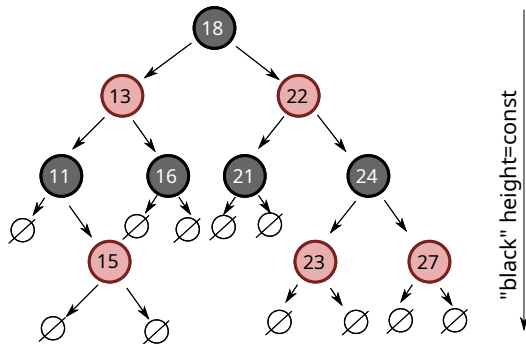
$x.\text{hashCode}() == y.\text{hashCode}() \not\Rightarrow x.\text{equals}(y)$

TreeMap

- Implemented by Red-Black tree.
- Keys are sorted.
- Quite good lookup: $O(\log n)$
- Quite good insert: $O(\log n)$
- Quite good delete: $O(\log n)$
- Memory consumption: $O(n)$.

Red-black tree

For each path count of black nodes ("black length") is constant



Hello
○○

Algorithmics
○○○○○○○○○○

List
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

Maps
○○○○●

Excercises
○○○○○○○○

Summary
○○○○○

HashMap

Algorithms visualised

<https://www.cs.usfca.edu/~galles/visualization/java/visualization.html>

How to write tests

- Create test that checks constraint on empty or broken code
- Run the test to confirm that it fails (if it passes, the checks are incorrect)
- During the test compilation it might be needed to create more empty methods - this is the Development part in TDD
- Fix the code and fill the empty methods
- Tests should pass

Exercise 1

- Open the project, and enter the `t1` package
- in `src/main/java/t1` we have code that we need to fix
- in `src/test/java/t1` we have tests that need to pass
- Please implement `FIFOQueue` using appropriate Java's list implementation
- Please implement `LIFOQueue` using appropriate Java's list implementation
- What happens when we change the selected implementation of list to another? Does the test still pass? What is the difference

Excercise 2

- see `src/main/java/t2` and `src/test/java/t2`
- First focus on `first_exercise_testGBPtoPLN`
- why there is no such currency conversion, providing that we have added it in the map in `initStatic()` method?
- Next focus on `second_exercise_testPLNtoGBP`
- Why the test is failing? What is the printed result?

Exercise 3

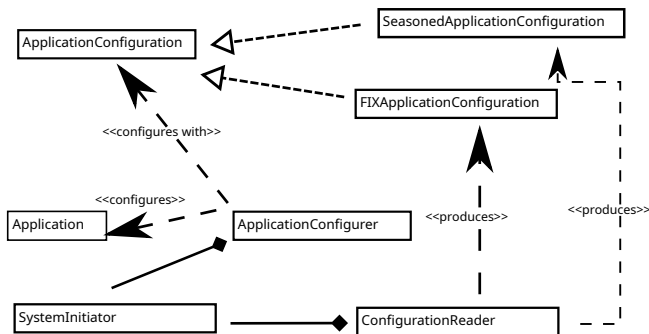
- see `src/main/java/t3` and `src/test/java/t3`
- List must not grow, must throw proper exceptions.
- List must contain the java's Array as storage for the items
- How to initialize the storing array for any T? Why `new T[]` is not working?

Excercise 4

- see `src/main/java/t4` and `src/test/java/t4`
- Why the `assertNull` does not work?

Excercise 5

- see `src/main/java/t5` and `src/test/java/t5`
- Uncomment code in the test
- Why the code does not compile? How can we fix it?



Exercise 6

- see `src/main/java/t6` and `src/test/java/t6`
- `testRunningTimeEasy` - focus on time of execution, why it is so slow
- `testMemEfficiencyEasy` - focus on time of execution, why it is so slow

Hello
oo

Algorithmics
oooooooo

List
oooooooooooooooooooooooo

Maps
ooooo

Exercises
ooooooo

Summary
●oooo

What have we learned today?

Summary

What have we learned today?

Test Driven Development

- Always start with test

What have we learned today?

Test Driven Development

- Always start with test
- Test must fail

What have we learned today?

Test Driven Development

- Always start with test
- Test must fail
- Make the test passing by fixing the code

What have we learned today?

Test Driven Development

- Always start with test
- Test must fail
- Make the test passing by fixing the code

What have we learned today?

Complexity as a gift from Java

- Big O notation, small o notation, theta θ notation

What have we learned today?

Complexity as a gift from Java

- Big O notation, small o notation, theta θ notation
- Java's *sophisticated* collections give us access to best algorithm implementation

What have we learned today?

Complexity as a gift from Java

- Big O notation, small o notation, theta θ notation
- Java's *sophisticated* collections give us access to best algorithm implementation
- Even though, using incorrect implementation, we can break things up

What have we learned today?

Complexity as a gift from Java

- Big O notation, small o notation, theta θ notation
- Java's *sophisticated* collections give us access to best algorithm implementation
- Even though, using incorrect implementation, we can break things up
- Even though, sometimes we may need to have our own implementation (read: **use libraries**)

What have we learned today?

Generics

- One code for many usages
- ? extends and ? super
- Type erasure will clear typed info

Hello

oo

Algorithmics

oooooooooo

List

oooooooooooooooooooooooooooo

Maps

ooooo

Excercises

ooooooo

Summary

oooo●

What have we learned today?

Thank you