

Laboratory 3, Collections

Main goal

Students will learn the main data structures of Java programming - collections - their use cases, performance issues and pitfalls.

Covered topics: Generic classes, wildcards, Generic methods, Primitive wrappers, Autoboxing, Collection interfaces, implementations, factory method.

Required for the lab

Java types and their sizes, Generic classes, creating, type erasure, sorting algorithms, $O(n)$ notation, autoboxing.

- `add()`, `get()`, `size()`, `remove()` of List interface, time and memory cost of `LinkedList` and `ArrayList` implementations
- `add()`, `offer()`, `poll()`, `peek()`, `remove()`, methods' contract of Queue interface
- `Integer.valueOf()` and `Boolean.valueOf()` implementation details
- please compile a class with `int`, `boolean` and `long` autoboxing and use `javap -c classname.class` to see how autoboxing is implemented
- type erasure
- (*) what is amortized time cost of `ArrayList.add(Object o)` operation (prove using coin method)

Plan

Students will be provided with `src/main` and `src/test` code which will contain gaps to fill and errors to correct. Students have to fix code while not changing the code of tests (unless explicitly asked to), to make the tests pass. This will need them to investigate the problem highlighted in test commentary, find appropriate fix, and rerun tests to make sure the fix was correct.

Way of getting a credit for the classes

Students will follow exercises given by teacher. There will be no crediting exam on this classes. There will be a project after this laboratory.

Literature

1. <http://docs.oracle.com/javase/tutorial/collections/>
2. <http://docs.oracle.com/javase/tutorial/java/generics/>
3. Provided source code.

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

Introduction

- We're going to have some fun with lists and maps
- Lists and maps are **basic** data structures, and in this context "Basic" means *being a base of everything else*

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

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

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

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:

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

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

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.

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

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:

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

Lists

Algorithmics

○○○○○●

List

○○○○○○○○○

Maps

○○○

Complexity calculation and nomenclature

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

Lists

Algorithms
○○○○●

List
○○○○○○○

Maps
○○○

Complexity calculation and nomenclature

Worst case scenario

Lists

Algorithms
○○○○○○

List
●○○○○○○○

Maps
○○○

ArrayList

Need to know by heart

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 $n \cdot \log n$

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

Lists

Algorithms
○○○○○○

List
●○○○○○○○

Maps
○○○

ArrayList

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**): $O(n)$

Lists

Algorithms
○○○○○○

List
●○○○○○○○

Maps
○○○

ArrayList

ArrayList

- Quick special-case insert, when for n equal number of elements, further referred to as **append**: $\Theta(1)$, 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 $\Theta(1)$???

Lists

Algorithms
○○○○○○

List
○○●○○○○○

Maps
○○○

ArrayList

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

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

Lists

Algorithms
○○○○○○

List
○○○○●○○○

Maps
○○○

ArrayList

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:

Lists
ArrayList

Algorithms
○○○○○

List
○○○○○●○○

Maps
○○○

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

Lists
ArrayList

Algorithms
○○○○○

List
○○○○○●○

Maps
○○○

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.

Lists
LinkedList

Algorithms
○○○○○

List
○○○○○○○●

Maps
○○○

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.

Lists
HashMap

Algorithms
○○○○○

List
○○○○○○○○○

Maps
●○○○

HashMap

- Fast finding element by the key (further referred to as: **lookup**): $O(1)$.¹
- 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.

¹When 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.

Lists
HashMap

Algorithms
○○○○○

List
○○○○○○○○○

Maps
○●○

HashMap

$x.equals(y) \implies x.hashCode() == y.hashCode()$
 $x.hashCode() == y.hashCode() \not\Rightarrow x.equals(y)$

Lists
HashMap

Algorithms
○○○○○

List
○○○○○○○○○

Maps
○○●

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: $O(n)$.