Gebze Technical University
Computer Engineering

CSE 222
2017 Spring

HOMEWORK 4 REPORT

VAKHID BETRAKHMADOV
141044086

Course Assistant: Nur Banu Albayrak

Analysis

Q1.

Stack is a well-known LIFO data structure. We started by writing a simple interface StackInterface for a stack which is supposed to be implemented in our all stack versions. First stack version, namely StackA, extends ArrayList and inherits all ArrayList's methods. It implements StackInterface's methods using technic known as delegation. When pushing to and popping from the StackA we add and remove elements from the end of the underling ArrayList, because both operations to the end of an ArrayList take constant time, in comparison to addition and removal from the head of an ArrayList, which takes liner time. As a huge minus for this implementation we can mark all those methods inherited from ArrayList, which breach the principles of a Stack, that is random access to the elements of a Stack and operations on them are not allowed. Addition and removal operations on a Stack are supposed to be strictly according to LIFO principle. We can achieve this only if we use StackA as a StackInterface, but that is no guaranteed.

In our second version StackB we use composition, and instead of extending ArrayList, we simply maintain a private object of an ArrayList. The same as in the StackA we still add and remove to the end in our push and pop operations using delegation. But this version is much better, then the first one, because we don't have the overhead of unnecessary methods, so our Stack principles are safe. Efficiency is the same. In the third version, StackC, we maintain our stack by means of internal nodes. Pushing to and popping new elements from the stack take constant time, because we always add and remove from the head. Stack principles are safe, but in comparison with StackA and StackB (in which capacity is doubled when size == capacity) this implementation might be a bit slower, because every addition of a new element requires new allocation and removal brings after itself garbage collection.

The worst version in terms of efficiency is the fourth one, StackD. We maintain our stack by means of a queue using composition and delegation technics. Stack and Queue are two opposite data structures. Addition and removal in a queue happens from the different ends, so in order to use queue as a stack we need to traverse the queue in either of operations (push or pop). In my implementation, I use ArrayDeque collection referenced by a reference of type Queue. Pushing operation in this implementation is still constant time, because of the addition to the end and underling ArrayList like structure of ArrayDeque. But popping operations in such implementation take liner time, because first we need to traverse the whole queue using iterator just to get to the end of the queue and remove last pushed element, this is very 'expensive'. Stack principles stay safe.

So, as a conclusion, as for me, the best version of all four different implementations is StackB. StackB guarantees Stack principles safety, it doesn't require any unnecessary garbage collection and too frequent memory allocation, neither it has any overhead of memory for maintenance of nodes references as in StackC if the stack is full.

Test cases:

1) Popping from an empty stack. Result: NoSuchElementException
2) Casting an instance of StackA down to ArrayList and acquiring random access to elements
3) Pushing a lot of elements to StackD and then popping them back.

Running command and results:

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...

Process finished with exit code 0
```

test.csv | testResult_1.csv

```
7,7,6,5,4,3,2,1
5,4.0,6.7,4.5,2.3,1.2
7,f,d,t,j,h,g,a
3,defter,kalem,kitap
7,7,6,5,4,3,2,1
5,4.0,6.7,4.5,2.3,1.2
7,f,d,t,j,h,g,a
3,defter,kalem,kitap
7,7,6,5,4,3,2,1
5,4.0,6.7,4.5,2.3,1.2
7,f,d,t,j,h,g,a
3,defter,kalem,kitap
7,7,6,5,4,3,2,1
5,4.0,6.7,4.5,2.3,1.2
7,f,d,t,j,h,g,a
3,defter,kalem,kitap
```

test.csv | testResult_2.csv

```
1  7,7,6,5,4,3,2,1
2  5,4.0,6.7,4.5,2.3,1.2
3  7,f,d,t,j,h,g,a
4  3,defter,kalem,kitap
5  7,7,6,5,4,3,2,1
6  5,4.0,6.7,4.5,2.3,1.2
7  7,f,d,t,j,h,g,a
8  3,defter,kalem,kitap
9
```

testResult_3.csv

```
1  7,1,2,3,4,5,6,7
2  5,1.2,2.3,4.0,4.5,6.7
3  7,a,d,f,g,h,j,t
4  3,defter,kalem,kitap
5  7,1,2,3,4,5,6,7
6  5,1.2,2.3,4.0,4.5,6.7
7  7,a,d,f,g,h,j,t
8  3,defter,kalem,kitap
9
```