

Exercises

W05H02 - PUM Server Upgrade

Points

0 / 10

Submission due date

in 4 days

Status

No graded result

Difficulty

Categories

Hausaufgabe

Tasks:

PUM Server Upgrade

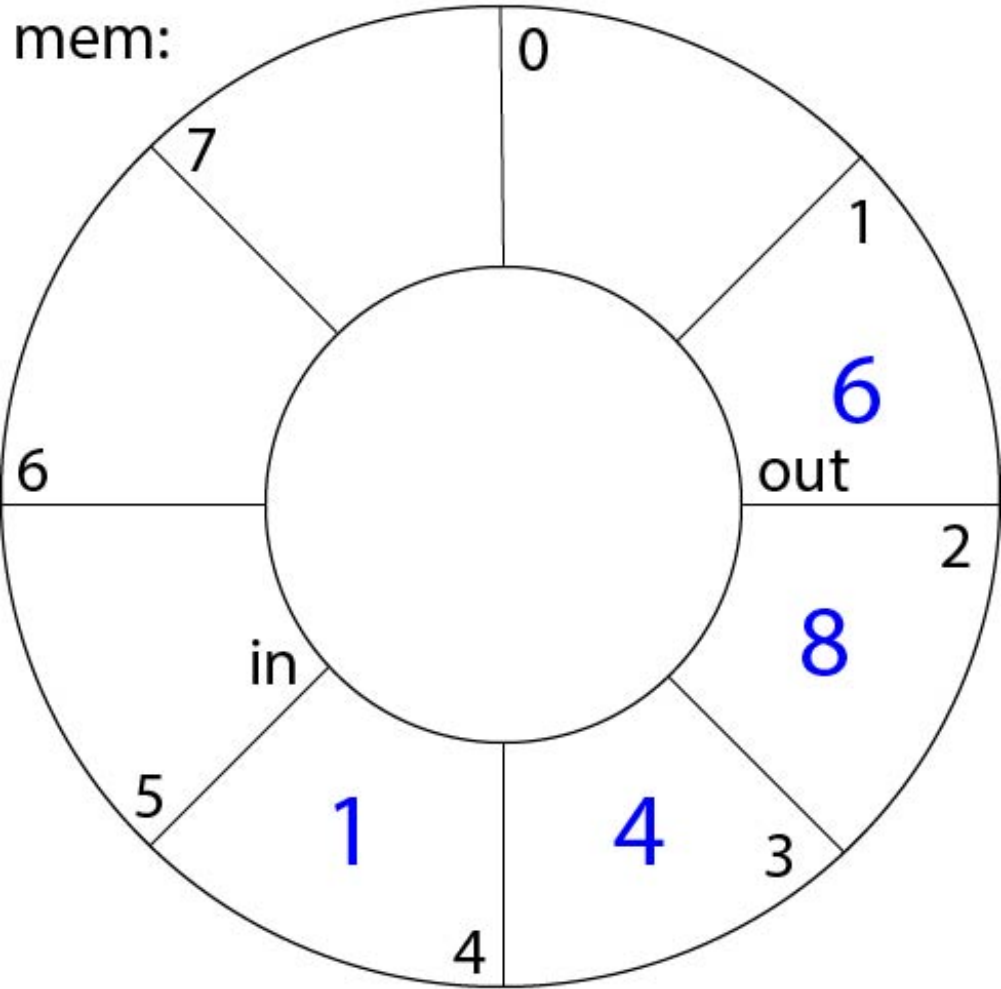
In dieser Aufgabe geht es um unterschiedliche Datenstrukturen wie RingBuffer und MultiStack zu implementieren.

Die Forschuine der PUM sind begeistert von unserem LRZ. Um noch besser forschen zu können, wollen sie nun auch einen eigenen Hochleistungsrechner bauen. Die Softwareuine wurden beauftragt, einige notwendige Komponenten zu implementieren. Als exzellenter Freund unterstützt du sie dabei.

RingBuffer

Um Daten asynchron zwischen den Server-Nodes hin und her zu senden, benötigen die Pinguine einen effizienten Buffer, genauer einen RingBuffer. Ein RingBuffer wird mit seiner Kapazität initialisiert (das ist die maximale Anzahl an Einträgen, die gespeichert werden kann). In den RingBuffer können dann Werte gelegt (put(int)) und aus ihm wieder entnommen (get()) werden.

Hier siehst du eine Visualisierung eines RingBuffers:



Wie du siehst, speichern wir die Werte in einem Array (mem). Um festzuhalten, in welchem Bereich valide Daten gespeichert sind, benutzen wir die beiden Felder in und out. in speichert dabei den Index, an dem der nächste Wert, der via put(int) eingefügt werden soll, in mem abgespeichert wird. out zeigt auf den Index, von dem bei der nächsten get()-Operation gelesen wird. Durch eine Sequenz von put- und get-Operationen kann dieser Bereich also durch das Array "wandern", auch über die Grenzen des Arrays hinaus. Dazu musst du dir vorstellen, dass das Array ein Ring ist. Wir interpretieren also mem[0] als Nachfolger von mem[mem.length-1] und mem[mem.length-1] als Vorgänger von mem[0].

Um dir die Idee hinter einem Ring Buffer besser vorstellen zu können, lohnt es sich, die Visualisierung im Wikipedia-Artikel anzuschauen. Die Felder in und out werden im Wikipedia-Artikel als write pointer bzw. als read pointer.

Die folgenden Teilaufgaben beschreiben die Implementierung step-by-step:

1. ☒ Als aller Erstes! No results

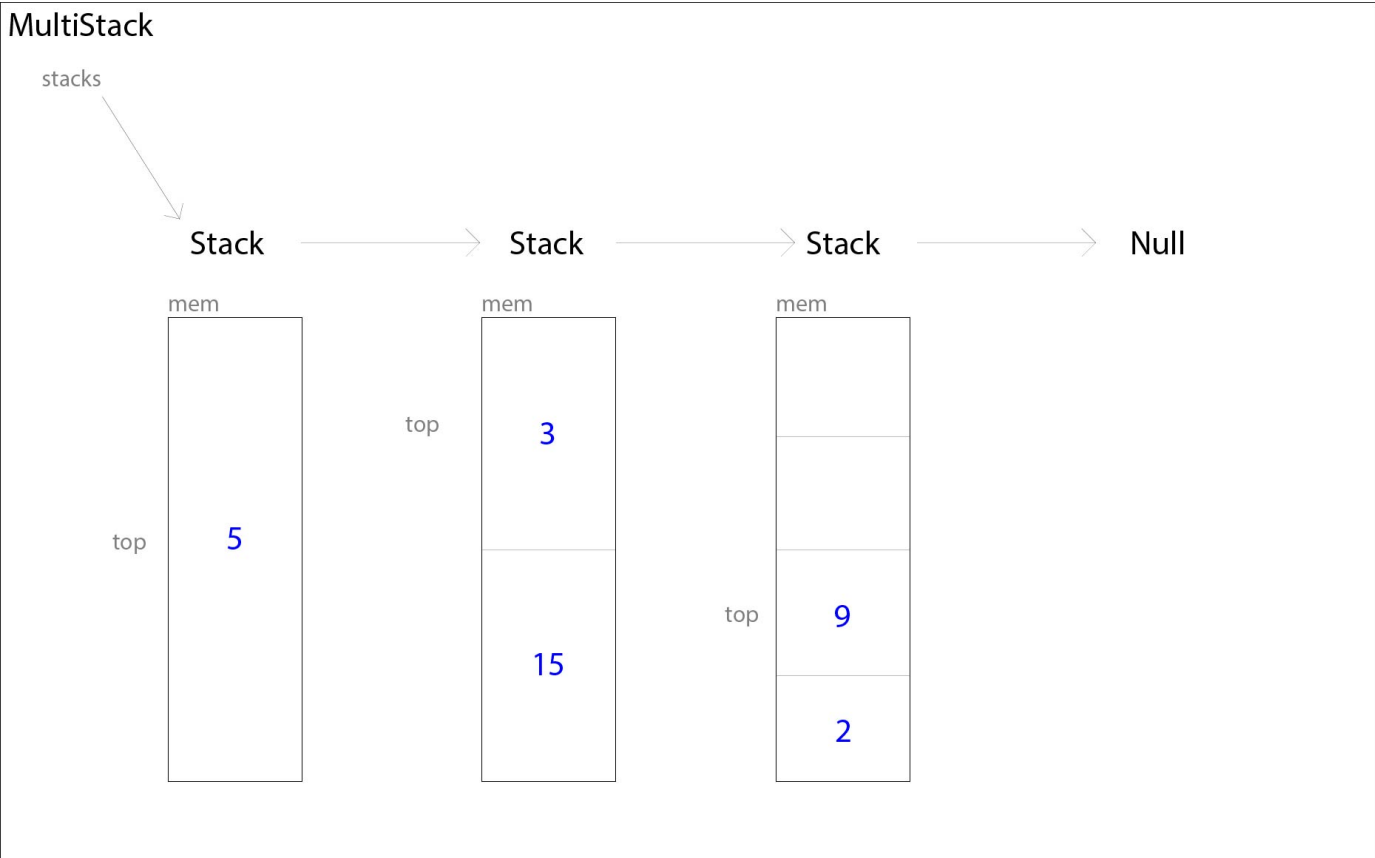
Lies dir zunächst die folgenden Teilaufgaben durch und erstelle erst die erfordernten Methoden, um sie anschließend zu implementieren. So hast du schonmal Code, den Artemis kompilieren und testen kann. Dies bezieht sich auch auf die Methoden, die in **MultiStack** implementiert werden müssen.

2. **Implementiere die Methode isEmpty!** No results
Erstelle die Methode `isEmpty`, die einen `boolean` zurückgeben soll: `true` - keine Einträge im `RingBuffer` gespeichert, `false` - einer oder mehr Einträge im `RingBuffer` gespeichert.
3. **Implementiere die Methode isFull!** No results
Erstelle die Methode `isFull`, die einen `boolean` zurückgeben soll: `true` - der `RingBuffer` speichert die maximale Anzahl von Einträgen, `false` - der `RingBuffer` speichert weniger als die maximale Anzahl von Einträgen.
4. **Implementiere die Methode put!** No results
Erstelle die Methode `put`, die einen `int` erwartet, um diesen im `RingBuffer` zu speichern. Sollte der Buffer voll sein, dürfen keine Werte in den Buffer eingefügt werden, stattdessen akzeptiert der `RingBuffer` den neuen Wert nicht. Ist das Speichern erfolgreich, soll der `boolean true` zurückgegeben werden, andernfalls `false`.
5. **Implementiere die Methode get!** No results
Erstelle die Methode `get`, die einen `int` zurückgeben soll. Wenn möglich soll der Wert zurück gegeben werden, der am längsten im `RingBuffer` liegt - dieser wird dann auch aus dem aktuellen Speicherbereich des Buffers entfernt - andernfalls erwarten wir den Default-Rückgabewert `Integer.MIN_VALUE`.
6. **Konstruktor & Performance!** No results
Der erste Test dient nur als Sicherheit, damit du sofort erkennst ob du ausversehen den Konstruktor verändert hast. Mach dir aber nicht zu viele Gedanken. Entscheidend ist nur, dass du nicht zu viele irrelevante Operationen ausführst (z.B. Kopien des Arrays erstellst).

MultiStack

Die Pinguine möchten auf ihrem HPC natürlich viele Java-Programme ausführen. Um komplexere Programme entwickeln zu können, benötigen sie einen Stack, der mehr Daten speichern kann. Ein erster Prototyp dafür soll in den folgenden Teilaufgaben entwickelt werden.

MultiStack nutzt die Klasse **Stack**, in der die Daten gespeichert werden. Jeder **Stack** ist dabei wie ein Listen-Element und enthält, abgesehen von dem Array `mem` (zum Speichern der Daten) und `top` (um auf den obersten Wert dieses **Stacks** zu zeigen), auch eine Referenz auf einen (möglichen) Nachfolger-**Stack**. Der Beginn dieser Liste von **Stacks** wird als Referenz `stacks` in **MultiStack** gespeichert.



1. **Als aller Erstes!** No results
Lies dir zunächst die folgenden Teilaufgaben durch und erstelle erst die erfordernten Methoden, um sie anschließend zu implementieren. So hast du schonmal Code, den Artemis kompilieren und testen kann.
2. **Implementiere die Methode push!** No results
Erstelle die Methode `push`, die einen `int` auf den **MultiStack** legen soll. Dabei soll wie folgt vorgegangen werden: Da der Wert in der Liste aus **Stacks** gespeichert wird, müssen wir zunächst den Ort finden, an dem der Wert abgelegt werden soll. Dazu starten wir bei dem ersten **Stack**. Ist dieser bereits gefüllt, führen wir unsere Suche im Nachfolger (`next`) fort bis wir uns im passenden **Stack** befinden. Sollten alle **Stacks** gefüllt sein, müssen wir einen neuen **Stack** mit der doppelten Kapazität des Vorgängers erstellen (Je weiter du durch die Liste iterierst, desto größer werden die Stacks - Faktor 2). Da wir nun den richtigen **Stack** gefunden oder initialisiert haben, können wir in

Verfahren kann am einfachsten rekursiv implementiert werden, d.h. der **MultiStack** ruft die gleichnamige rekursive Methode in **Stack** auf. (Rekursion ist hier aber nicht unbedingt verlangt.)

3. **🕒 Implementiere die Methode top!** No results

Erstelle die Methode **top**, die den Wert (**int**) zurückgeben soll, der ganz oben auf dem **MultiStack** liegt. (Das ist der Wert, der als letztes eingefügt wurde.) Der Wert soll dabei *NICHT* entfernt werden. Ist der Stack leer, erwarten wir den Default-Rückgabewert **Integer.MIN_VALUE**. *Tipp:* Auch diese Teilaufgabe lässt sich sehr einfach rekursiv lösen.

4. **🕒 Implementiere die Methode pop!** No results

Erstelle die Methode **pop**, die den Wert (**int**) zurückgeben *UND* entfernen soll, der ganz oben auf dem **MultiStack** liegt. (Das ist der Wert der als letztes eingefügt wurde.) Ist der letzte Stapel der **Stack**-Liste leer, soll dieser aus der Liste gelöscht werden. Ausgenommen ist davon der erste Stack mit Kapazität **1**. Dieser soll immer erhalten bleiben, auch wenn er aktuell keine Werte speichert. (Mach dir gerne Gedanken darüber, warum.) Ist der Stack leer, erwarten wir den Default-Rückgabewert **Integer.MIN_VALUE**. *Tipp:* Wie du dir bestimmt denken kannst, kannst du auch diese Teilaufgabe durch eine rekursive Methode in **Stack** lösen.

5. **🕒 Implementiere den Konstruktor von Multistack!** No results

Der Test dient nur als Sicherheit, damit du sofort erkennst ob du ausversehen den Konstruktor verändert hast.

Viel Erfolg!

Exercise details

Release date	Nov 21, 2025 17:00
Submission due date	Nov 30, 2025 17:00
Complaint possible	No