Pfichtaufgabe 9 – Collections-Klassen

9.1 Collections-Klassen

Hier vergleichen Sie das Laufzeitverhalten einiger Datenstrukturen aus der Collections-API miteinander. So bekommen Sie ein Gefühl für die unterschiedlichen Zeitaufwände beim Lesen, Schreiben und Suchen. Beachten Sie aber bitte, dass Sie diese Zeiten nicht als absolute Werte betrachten. Je häufiger die Code-Zeilen durchlaufen werden, desto größer die die Wahrscheinlichkeit, dass der der Hotspot-Compiler den Code zur Laufzeit compiliert. Verwenden Sie für Ihre Lösung immer Datenstrukturen mit `int`-Werten und nutzen Sie in Ihrem Code aus, dass diese Datenstrukturen gemeinsame Schnittstellen implementieren. Mit der Arbeit auf Schnittstellen sparen Sie sehr viel Code ein. Da Sie `int`-Werte nicht direkt in den Datenstrukturen speichern können, müssen Sie die entsprechende Wrapper-Klasse `Integer` verwenden. Führen Sie die in folgender Tabelle genannten Messungen durch.

Operation	**Datenstrukturen**
Hängen Sie an die leere Datenstruktur nacheinander die `int`-Zahlen von `0` bis `99.999` an. Prüfen Sie, ob bei `Vector` und `ArrayList` Geschwindigkeitssteigerungen feststellbar sind, wenn Sie beiden beim Erzeugen im Konstruktor eine Größe vorgeben.	`Vector`, `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`
Fügen Sie in die leere Datenstruktur nacheinander die `int`-Zahlen von `0` bis `99.999` immer am Anfang ein. Prüfen Sie, ob bei `Vector` und `ArrayList` Geschwindigkeitssteigerungen feststellbar sind, wenn Sie beiden beim Erzeugen im Konstruktor eine Größe vorgeben.	`Vector`, `ArrayList`, `LinkedList`
Nehmen Sie die oben gefüllten Datenstrukturen und suchen Sie mit Hilfe von Iteratoren den zuletzt eingefügten Wert.	`Vector`, `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`
Nehmen Sie die oben gefüllten Datenstrukturen und suchen Sie mit Hilfe der Binärsuche den zuletzt eingefügten Wert. Die Implementierung der Binärsuche finden Sie in der Klasse `Collection`.	`Vector`, `ArrayList`
Nehmen Sie die oben gefüllten Datenstrukturen und suchen Sie mit Hilfe der in den Datenstrukturen vorhandenen Methoden den zuletzt eingefügten Wert.	`Vector`, `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`

Nehmen der Messungen der Laufzeiten mit unterschiedlichen Datenstrukturen sollen Sie auch ermitteln, inwiefern sich die Zeitaufwände zwischen Iteratoren und Stream-Klassen unterscheiden. Führen Sie dazu die Messungen aus der folgenden Tabelle durch.

Operation	**Zugriffstechniken**
Erzeugen Sie eine `ArrayList` mit `int`-Zufallszahlen im Wertebereich zwischen 0 und 9999.	`add`-Methode der `ArrayList`, `generate`-Methode der Stream-API
Addieren Sie alle geraden Zufallszahlen aus der `ArrayList` des ersten Tests.	`Iterator`-Durchlauf, sequentieller Stream (erzeugt mit der Methode `stream` der `ArrayList`), möglicherweise paralleler Stream[1] (Methode `parallelStream` der `ArrayList`)

Wenn Sie einen sehr schnellen oder sehr langsamen Computer haben, dann müssen Sie in den oben genannten Punkten die Anzahl der einzufügenden Daten eventuell anpassen, damit Sie einerseits aussagekräftige Zahlen bekommen und andererseits nicht ewig warten müssen. Lassen Sie Ihre Methoden im selben Programmlauf mehrfach nacheinander laufen, um Schwankungen durch den Einsatz des JIT-Compilers auszugleichen. Implementierung der Zeitmessung:

```
public void methodeXY() {
  // Startzeit holen
  long start = System.currentTimeMillis();
  // Algorithmus laufen lassen
  // ...
  // Endzeit holen
  long end = System.currentTimeMillis();
  // Zeit in Millisekunden
  long durationInMsec = end - start;
}
```