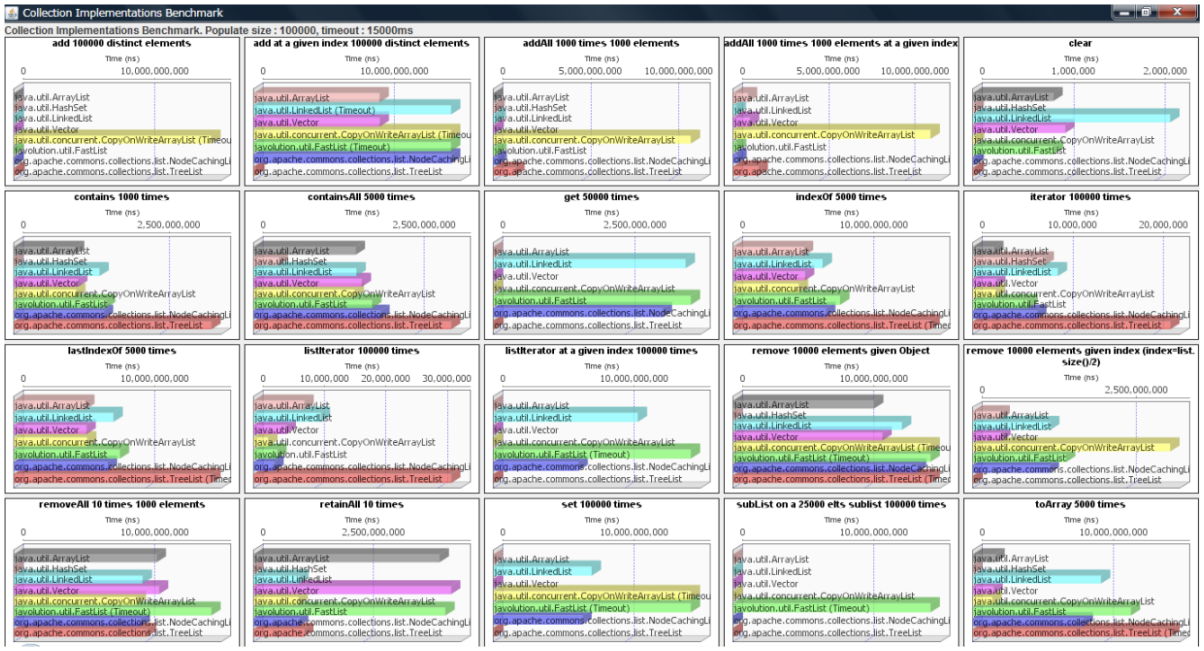
Collections

1. ArrayList oder LinkedList – wann nehmen Sie was?
Die ArrayList-Klasse ist die bekannteste und häufig die effizienteste List-Implementierung. ArrayList ist gut geeignet, wenn eine Liste im wesentlichen wie ein Array verwendet wird, man sich aber nicht auf eine bestimmte Array-Größe festlegen möchte.

Beim nachträgliche Einfügen/Entfernen von Elementen ist die LinkedList-Klasse besonders effizient durch Links. Wenn das häufig genutzt wird, ist eine LinkedList zu empfehlen. LinkedList implementiert zudem Queue und Deque. Somit wird eine angenehme Nutzung der Klasse als FIFO- und LIFO-Speicher ermöglicht.

2. Interpretieren Sie die Benchmarkdaten von:
http://java.dzone.com/articles/java-collection-performance. Fällt etwas auf?



Wir können hier einige interessante Dinge sehen; wir bestätigen, was wir bereits über einige Sammlungen wussten, das CopyOnWriteArray ist signifikant langsam bei Datenänderungen, usw.
Wir können auch einige signifikante Unterschiede zwischen ArrayList und LinkedList sehen (OK, der Benchmark deckt nicht alle Fälle ab, und vielleicht sind wir in dem speziellen Fall, in dem ArrayList besser ist als LinkedList. Wie auch immer, es gibt uns insgesamt einen interessanten Einblick).

3. Wieso ist CopyOnWriteArrayList scheinbar so langsam?
Beim Löschen von Elementen/Hinzufügen von Elementen zum Container wird ein neues Array erstellt. Anstatt also Elemente direkt zum alten Container hinzuzufügen, kopiert er die Elemente im alten Container in den neuen Container. Das ist zwar sicher, nimmt aber viel Zeit in Anspruch.

4. Wie erzeugen Sie eine thread-safe Collection (die sicher bei Nebenläufigkeit ist) (WAS?? die Arraylists, Linkedlists, Maps etc. sind NICHT sicher bei multithreading??? Wer macht denn so einen Mist???)
Ausschnitt aus der Implementierung, der zunächst einmal nur eine lesende Methode, nämlich die get()-Methode, und die relevanten Felder der CopyOnWriteArrayList zeigt:

```
public class CopyOnWriteArrayList<E> implements List<E> {
      private volatile transient Object[] array;
      private final Object[] getArray() {
      return array;
      }
      public E get(int index) {
      return (E)(getArray()[index]);
      }
}
```

5. Achtung Falle!

List|<Integer> list = new ArrayList<Integer>;

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
  int i = itr.next();
  if (i > 5) { // filter all ints bigger than 5
    list.remove();
  }
}
```

Falls es nicht klickt: einfach ausprobieren...
Macht das Verhalten von Java hier Sinn?
Gibt es etwas ähnliches bei Datenbanken? (Stichwort: Cursor. Ist der ähnlich zu Iterator?)

6. Nochmal Achtung Falle: What is the difference between get() and remove() with respect to Garbage Collection?
Mit get() kann man gezielt ein Element an einer bestimmten Position auslesen. Es müssen also nicht mehr alle Elemente mit eine Schleife durchlaufen werden.

Mit remove() wird das Element an der angegebenen Position in dieser Liste entfernt. Verschiebt alle nachfolgenden Elemente nach links (subtrahiert eins von deren Indizes).

7. Ihr neuer Laptop hat jetzt 8 cores! Ihr Code für die Verarbeitung der Elemente einer Collection sieht so aus:

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
   int i = itr.next();
   //do something with i….
}
```

War der Laptop eine gute Investition?
Aktuell: Rausgeschmissene Kohle
Warum? Nur 1 Core wird genutzt, die anderen 7 Cores sind am Däumchen drehen.

Für die Mutigen: mal nach map/reduce googeln!
Map/reduce wurde von Google entwickelt. Mit diesem Verfahren lassen sich große (un)strukturierte Datenmengen schnell verarbeiten. Diese Schnelligkeit wird gewonnen durch die Parallelisirung und Verteilung von Aufgaben. Dadurch müssen auch die anderen Cores arbeiten.