Performance Analysis of Java Collections

Java collection and map implementations performance comparison

Introduction

The project focused on testing the time taken to execute operations like ADD, CONTAINS, REMOVE, and CLEAR on Java collection and map interface implementations such as List, Queue, Set, and Map... . The results obtained in tests will be used to choose a suitable collection for a specific user case.

Program Design

PerfomanceTesting Class

The performance testing class is the driver class of this project.
Set, List, Queue, and Map interfaces are implemented with the corresponding child classes such as HashSet, TreeSet, LinkedHashSet, ArrayList, LinkedList, ArrayList, LinkedList, ArrayDequeue, PriorityQueue, HashMap, TreeMap, LinkedHashMap accordingly.

First, each collection is loaded with 100,000 integer objects with random values between 0 and 99,999.

The program contains 4 more classes MeasureAdd, MeasureContains, MeasureRemove, and MeasureClear which were built using generics to check the performances of Add, Contains, Remove, and Clear methods accordingly.

Measuring Methods of each class above mention will be called from the Performance Testing class with loaded collections.

MeasureAdd Class

This class provides methods to calculate the time taken to execute add() operation on collections.

Method Summary

Modifier	Type	Method Signature
public	long	`measureAddOperation(Set<Integer> set)`
		`measureAddOperation(List<Integer> list)`
		`measureAddOperation(Queue<Integer> queue)`
		`measureAddOperation(Map<Integer,Integer> map)`

Method Detail

measureAddOperation is overloaded to get a Set or List or Queue or Map Interface implementation. It returns the average time taken in nanoseconds to add a random element to a random index of Collection or map implementations by repeating the operation 100 times.

MeasureContains Class

This class provides methods to calculate the time taken to execute contains() operation on collections.

Method Summary

Modifier	Type	Method Signature
public	long	`measureContainsOperation(Set<Integer> set)`
		`measureContainsOperation(List<Integer> list)`

		measureContainsOperation(Queue<Integer> queue)
		measureContainsOperation(Map<Integer,Integer > map)

Method Detail

measureContainsOperation is overloaded to get a Set or List or Queue or Map Interface implementation. It returns the average time taken in nanoseconds to check the availability of a random element in Collection or map implementations by repeating the operation 100 times.

MeasureRemove Class

This class provides methods to calculate the time taken to execute remove() operation on collections.

Method Summary

Modifier	Type	Method Signature
public	long	measureRemoveOperation(Set<Integer> set)
		measureRemoveOperation(List<Integer> list)
		measureRemoveOperation(Queue<Integer> queue)
		measureRemoveOperation(Map<Integer,Integer> map)

Method Detail

measureRemoveOperation is overloaded to get a Set or List or Queue or Map Interface implementation. It returns the average time taken in nanoseconds to remove a random element from a random index of Collection or map implementations by repeating the operation 100 times.

MeasureClear Class

This class provides methods to calculate the time taken to execute clear() operation on collections.

Method Summary

Modifier	Type	Method Signature
public	long	`measureClearOperation(Set<Integer> set)`
		`measureClearOperation(List<Integer> list)`
		`measureClearOperation(Queue<Integer> queue)`
		`measureClearOperation(Map<Integer,Integer> map)`

Method Detail

measureClearOperation is overloaded to get a Set or List or Queue or Map Interface implementation. It returns the average time taken in nanoseconds to clear the whole Collection or map implementations by repeating the operation 100 times.

Dependencies

The nanoTime() method in the java.lang.System class is used to calculate the execution time of each operation in above 4 classes.

Full Java program code used for testing

PerfomanceTesting Class

```
import java.util.*;

public class PerfomanceTesting {
   public static void main(String[] args) {

        Set<Integer> hashSet = new HashSet<>();
        Set<Integer> treeSet = new TreeSet<>();
        Set<Integer> linkedHashSet = new LinkedHashSet<>();

        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();

        Queue<Integer> arrayDeque = new ArrayDeque<>();
        Queue<Integer> priorityQueue = new PriorityQueue<>();

        Map<Integer, Integer> hashMap = new HashMap<>();
        Map<Integer, Integer> treeMap = new TreeMap<>();
        Map<Integer, Integer> linkedHashMap = new LinkedHashMap<>();

        Random random = new Random();
        for (int i = 0; i < 100000; i++) {
            int randomValue = random.nextInt(100000);
            hashSet.add(randomValue);
            treeSet.add(randomValue);
            linkedHashSet.add(randomValue);
            arrayList.add(randomValue);
            linkedList.add(randomValue);
```

```
            arrayDeque.add(randomValue);
            priorityQueue.add(randomValue);
            hashMap.put(i, randomValue);
            treeMap.put(i, randomValue);
            linkedHashMap.put(i, randomValue);
        }

        MeasureAdd add_test = new MeasureAdd();

        long hashSet_addTime = add_test.measureAddOperation(hashSet);
        System.out.println("HashSet add       : " + hashSet_addTime + "
nanoseconds");

        long treeSet_addTime = add_test.measureAddOperation(treeSet);
        System.out.println("TreeSet add       : " + treeSet_addTime + "
nanoseconds");

        long linkedHashSet_addTime =
add_test.measureAddOperation(linkedHashSet);
        System.out.println("LinkedHashSet add : " + linkedHashSet_addTime + "
nanoseconds");

        long arrayList_addTime = add_test.measureAddOperation(arrayList);
        System.out.println("ArrayList add     : " + arrayList_addTime + "
nanoseconds");

        long linkedList_addTime = add_test.measureAddOperation(linkedList);
        System.out.println("LinkedList add    : " + linkedList_addTime + "
nanoseconds");

        long arrayDeque_addTime = add_test.measureAddOperation(arrayDeque);
        System.out.println("ArrayDeque add    : " + arrayDeque_addTime + "
nanoseconds");

        long priorityQueue_addTime =
add_test.measureAddOperation(priorityQueue);
        System.out.println("PriorityQueue add : " + priorityQueue_addTime + "
nanoseconds");

        long hashMap_addTime = add_test.measureAddOperation(hashMap);
        System.out.println("HashMap add       : " + hashMap_addTime + "
nanoseconds");

        long treeMap_addTime = add_test.measureAddOperation(treeMap);
        System.out.println("TreeMap add       : " + treeMap_addTime + "
nanoseconds");

        long linkedHashMap_addTime =
add_test.measureAddOperation(linkedHashMap);
```

```
        System.out.println("LinkedHashMap add : " + linkedHashMap_addTime + "
nanoseconds");



        MeasureContains containsTest = new MeasureContains();

        long hashSet_containsTime =
containsTest.measureContainOperation(hashSet);
        System.out.println("HashSet contains       : " + hashSet_containsTime
+ " nanoseconds");

        long treeSet_containsTime =
containsTest.measureContainOperation(treeSet);
        System.out.println("TreeSet contains       : " + treeSet_containsTime
+ " nanoseconds");

        long linkedHashSet_containsTime =
containsTest.measureContainOperation(linkedHashSet);
        System.out.println("LinkedHashSet contains : " +
linkedHashSet_containsTime + " nanoseconds");

        long arrayList_containsTime =
containsTest.measureContainOperation(arrayList);
        System.out.println("ArrayList contains     : " +
arrayList_containsTime + " nanoseconds");

        long linkedList_containsTime =
containsTest.measureContainOperation(linkedList);
        System.out.println("LinkedList contains    : " +
linkedList_containsTime + " nanoseconds");

        long arrayDeque_containsTime =
containsTest.measureContainOperation(arrayDeque);
        System.out.println("ArrayDeque contains    : " +
arrayDeque_containsTime + " nanoseconds");

        long priorityQueue_containsTime =
containsTest.measureContainOperation(priorityQueue);
        System.out.println("PriorityQueue contains : " +
priorityQueue_containsTime + " nanoseconds");

        long hashMap_containsTime =
containsTest.measureContainOperation(hashMap);
        System.out.println("HashMap contains       : " + hashMap_containsTime
+ " nanoseconds");
```

```
	long treeMap_containsTime =
containsTest.measureContainOperation(treeMap);
	System.out.println("TreeMap contains       : " + treeMap_containsTime
+ " nanoseconds");

	long linkedHashMap_containsTime =
containsTest.measureContainOperation(linkedHashMap);
	System.out.println("LinkedHashMap contains : " +
linkedHashMap_containsTime + " nanoseconds");



	MeasureRemove removeTest = new MeasureRemove();

	long hashSet_removeTime = removeTest.measureRemoveOperation(hashSet);
	System.out.println("HashSet remove       : " + hashSet_removeTime + "
nanoseconds");

	long treeSet_removeTime = removeTest.measureRemoveOperation(treeSet);
	System.out.println("TreeSet remove       : " + treeSet_removeTime + "
nanoseconds");

	long linkedHashSet_removeTime =
removeTest.measureRemoveOperation(linkedHashSet);
	System.out.println("LinkedHashSet remove : " +
linkedHashSet_removeTime + " nanoseconds");

	long arrayList_removeTime =
removeTest.measureRemoveOperation(arrayList);
	System.out.println("ArrayList remove     : " + arrayList_removeTime +
" nanoseconds");

	long linkedList_removeTime =
removeTest.measureRemoveOperation(linkedList);
	System.out.println("LinkedList remove    : " + linkedList_removeTime +
" nanoseconds");

	long arrayDeque_removeTime =
removeTest.measureRemoveOperation(arrayDeque);
	System.out.println("ArrayDeque remove    : " + arrayDeque_removeTime +
" nanoseconds");

	long priorityQueue_removeTime =
removeTest.measureRemoveOperation(priorityQueue);
	System.out.println("PriorityQueue remove : " +
priorityQueue_removeTime + " nanoseconds");

	long hashMap_removeTime = removeTest.measureRemoveOperation(hashMap);
```

```
        System.out.println("HashMap remove       : " + hashMap_removeTime + "
nanoseconds");

        long treeMap_removeTime = removeTest.measureRemoveOperation(treeMap);
        System.out.println("TreeMap remove       : " + treeMap_removeTime + "
nanoseconds");

        long linkedHashMap_removeTime =
removeTest.measureRemoveOperation(linkedHashMap);
        System.out.println("LinkedHashMap remove : " +
linkedHashMap_removeTime + " nanoseconds");


        MeasureClear clearTest = new MeasureClear();

        long hashSet_clearTime = clearTest.measureClearOperation(hashSet);
        System.out.println("HashSet clear       : " + hashSet_clearTime + "
nanoseconds");

        long treeSet_clearTime = clearTest.measureClearOperation(treeSet);
        System.out.println("TreeSet clear       : " + treeSet_clearTime + "
nanoseconds");

        long linkedHashSet_clearTime =
clearTest.measureClearOperation(linkedHashSet);
        System.out.println("LinkedHashSet clear : " + linkedHashSet_clearTime
+ " nanoseconds");

        long arrayList_clearTime = clearTest.measureClearOperation(arrayList);
        System.out.println("ArrayList clear     : " + arrayList_clearTime + "
nanoseconds");

        long linkedList_clearTime =
clearTest.measureClearOperation(linkedList);
        System.out.println("LinkedList clear    : " + linkedList_clearTime + "
nanoseconds");

        long arrayDeque_clearTime =
clearTest.measureClearOperation(arrayDeque);
        System.out.println("ArrayDeque clear    : " + arrayDeque_clearTime + "
nanoseconds");

        long priorityQueue_clearTime =
clearTest.measureClearOperation(priorityQueue);
        System.out.println("PriorityQueue clear : " + priorityQueue_clearTime
+ " nanoseconds");

        long hashMap_clearTime = clearTest.measureClearOperation(hashMap);
```

```
        System.out.println("HashMap clear       : " + hashMap_clearTime + "
nanoseconds");

        long treeMap_clearTime = clearTest.measureClearOperation(treeMap);
        System.out.println("TreeMap clear       : " + treeMap_clearTime + "
nanoseconds");

        long linkedHashMap_clearTime =
clearTest.measureClearOperation(linkedHashMap);
        System.out.println("LinkedHashMap clear : " + linkedHashMap_clearTime
+ " nanoseconds");

    }
}
```

MeasureAdd Class

```
import java.util.*;
public class MeasureAdd {
    //Sets
    public long measureAddOperation(Set<Integer> set) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer elementToAdd = random.nextInt(100000);

            startTime = System.nanoTime();
            set.add(elementToAdd);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            set.remove(elementToAdd);
        }
        return (totalTime /100);
    }
    //List
    public long measureAddOperation(List<Integer> list) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer elementToAdd = random.nextInt(100000);
            Integer index = random.nextInt(100000);
```

```
            startTime = System.nanoTime();
            list.add(index,elementToAdd);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            list.remove(elementToAdd);
        }
        return (totalTime /100);
    }
    //Queue
    public long measureAddOperation(Queue<Integer> queue) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer elementToAdd = random.nextInt(100000);

            startTime = System.nanoTime();
            queue.add(elementToAdd);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            queue.remove(elementToAdd);
        }
        return (totalTime /100);
    }
    //Map
    public long measureAddOperation(Map<Integer,Integer> map) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer elementToAdd = random.nextInt(100000);
            Integer index = random.nextInt(100000);

            startTime = System.nanoTime();
            map.put(index,elementToAdd);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            map.remove(elementToAdd);
        }
        return (totalTime /100);
    }
}
```

MeasureContains Class

```
import java.util.*;


public class MeasureContains {

   public long measureContainOperation(Set<Integer> set) {
       long startTime, endTime, totalTime = 0;
       Random random = new Random();

       for (int i = 0; i < 100; i++) {
           Integer value = random.nextInt(100000);

           startTime = System.nanoTime();
           set.contains(value);
           endTime = System.nanoTime();
           totalTime += (endTime - startTime);

       }
       return (totalTime /100);
   }
   //List
   public long measureContainOperation(List<Integer> list) {
       long startTime, endTime, totalTime = 0;
       Random random = new Random();

       for (int i = 0; i < 100; i++) {
           Integer value = random.nextInt(10000);

           startTime = System.nanoTime();
           list.contains(value);
           endTime = System.nanoTime();
           totalTime += (endTime - startTime);

       }
       return (totalTime /100);
   }
   //Queue
   public long measureContainOperation(Queue<Integer> queue) {
       long startTime, endTime, totalTime = 0;
       Random random = new Random();
```

```
        for (int i = 0; i < 100; i++) {
            Integer value = random.nextInt(100000);

            startTime = System.nanoTime();
            queue.contains(value);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

        }
        return (totalTime /100);
    }
    //Map
    public long measureContainOperation(Map<Integer,Integer> map) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer value = random.nextInt(100000);
            startTime = System.nanoTime();
            map.containsValue(value);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);
        }
        return (totalTime /100);
    }
}
```

MeasureRemove Class

```
import java.util.*;

public class MeasureRemove {
    // Set
    public long measureRemoveOperation(Set<Integer> set) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
            Integer elementToRemove = random.nextInt(100000);

            startTime = System.nanoTime();
            set.remove(elementToRemove);
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            if(set.size() != 100000){
                Integer elementToAdd = random.nextInt(100000);
                set.add(elementToAdd);
            }
```

```
    }
    return (totalTime / 100);
}

// List
public long measureRemoveOperation(List<Integer> list) {
    long startTime, endTime, totalTime = 0;
    Random random = new Random();

    for (int i = 0; i < 100; i++) {
        Integer elementToRemove = random.nextInt(100000);

        startTime = System.nanoTime();
        list.remove(elementToRemove); // Remove by value
        endTime = System.nanoTime();
        totalTime += (endTime - startTime);

        if(list.size() != 100000) {
            Integer elementToAdd = random.nextInt(100000);
            list.add(elementToAdd);
        }

    }
    return (totalTime / 100);
}

// Queue
public long measureRemoveOperation(Queue<Integer> queue) {
    long startTime, endTime, totalTime = 0;
    Random random = new Random();

    for (int i = 0; i < 100; i++) {
        startTime = System.nanoTime();
        queue.remove(); // Remove from the head of the queue
        endTime = System.nanoTime();
        totalTime += (endTime - startTime);

        Integer elementToAdd = random.nextInt(100000);
        queue.add(elementToAdd);

    }
    return (totalTime / 100);
}

// Map
public long measureRemoveOperation(Map<Integer, Integer> map) {
    long startTime, endTime, totalTime = 0;
    Random random = new Random();
```

```
        for (int i = 0; i < 100; i++) {
            Integer index = random.nextInt(100000);
            Integer element = random.nextInt(100000);

            startTime = System.nanoTime();
            map.remove(element); // Remove by value
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            if(!map.containsValue(element)){
                map.put(index,element);
            }
        }
        return (totalTime / 100);
    }

}
```

MeasureClear Class

```
import java.util.*;
public class MeasureClear {
    // Sets
    public long measureClearOperation(Set<Integer> set) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {

            startTime = System.nanoTime();
            set.clear();
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            for (int j = 0; j < 100000; j++) {
                set.add(random.nextInt(100000));
            }
        }
        return (totalTime / 100);
    }

    // Lists
    public long measureClearOperation(List<Integer> list) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {
```

```
            startTime = System.nanoTime();
            list.clear();
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            for (int j = 0; j < 100000; j++) {
                list.add(random.nextInt(100000));
            }
        }
        return (totalTime / 100);
    }

    // Maps
    public long measureClearOperation(Map<Integer, Integer> map) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i < 100; i++) {

            startTime = System.nanoTime();
            map.clear();
            endTime = System.nanoTime();
            totalTime += (endTime - startTime);

            for (int j = 0; j < 100000; j++) {
                map.put(random.nextInt(100000), random.nextInt(100000));
            }
        }
        return (totalTime / 100);
    }

    public long measureClearOperation(Queue<Integer> queue) {
        long startTime, endTime, totalTime = 0;
        Random random = new Random();

        for (int i = 0; i<100; i++){
            startTime = System.nanoTime();
            queue.clear();
            endTime = System.nanoTime();
            totalTime += (endTime- startTime);

            for(int j = 0 ; j < 100000; j++){
                queue.add(random.nextInt(100000));
            }
        }
        return (totalTime/100);
    }
}
```

Comparison table of performance data

	Add(ns)	Contains(ns)	Remove(ns)	Clear(ns)
HashSet	295	569	454	37484
TreeSet	1010	891	1248	3608
LinkedHashSet	295	297	252	29536
ArrayList	14042	179014	179832	33424
LinkedList	408753	747282	695843	350741
ArrayDeque	377	191966	216	57967
PriorityQueue	202	229559	2004	37183
HashMap	368	471778	618	72939
TreeMap	855	1300722	4793	284
LinkedHashMap	370	849599	1068	63500

Discussion

Discussion is done according to the data obtained by above comprehensive performance analysis. The results, as depicted in the above table, indicate significant variations in performance times across different operations and

data structures. These variations can be attributed to the inherited characteristics and underlying algorithms.

When considering the first method Add, LinkedList took the highest execution time. Since we add an element to a random index of the collection rather than front or end and the size of the collection is x100000, LinkedList has an overhead of node allocations and ArrayList has an overhead of shifting elements. so these two take higher execution times.
Queue uses a FIFO approach so it always adds an element to its tail and has no such overheads mentioned in LinkedLists and ArrayLists. Therefore Queue implementations are much faster in case of adding an element whilst set and map implementations remain in between.

The contains method is used to see if the object exists rather than index-based accessing.
Contains method executed fastest in LinkedHashSet while the slowest was TreeMap.In ideal HashSet implementations have a time complexity of O(1) for the Contains method. But it is more likely to occur hashcollisions which decreases the performance. In spite of those, HashSets are widely used for lookups(i.e. Contains). So set implementations take less time to execute contains methods.
Since we used the containsValue method in the case of maps, there's nothing the HashMap can do but check all values and see if they're equal to the one we're searching. Therefore, the time complexity is O(n) with n being the number of elements in the HashMap not the size of the map. Thus above data show the slowest execution of the contains operation but if we used the key to access the above execution times would have been much less.

Linked list had the slowest element removal because we are removing an element in somewhere middle, it has to be found first. So its time complexity is in O(n) while removing from the front and end is O(1). Queues, Maps, and Sets have lesser execution times compared to Lists.

The TreeMap clear method was executed fastest according to the above table. In general, most of the implementations' execution times are in Order of n with n being the number of elements in the collection. As seen in the above table, Maps, Queues, Sets, and Lists take approximately a similar

time to execute clear method whilst LinkedList has a slightly higher execution time.

Conclusion

The performance of add(), contains(), remove(), and clear() methods in Java's Collection framework and Map interface can depend on various factors such as the Java Virtual Machine (JVM) implementation, the specific hardware it's running on, and the state of the JVM at the time of execution.

So, the above data can't be expected in all the cases.
The above discussion was done according to the data we've obtained but it is very likely to be different for someone else's test.

Therefore choosing a suitable Java collection or map highly depends on the use case and the which operations to be performed.