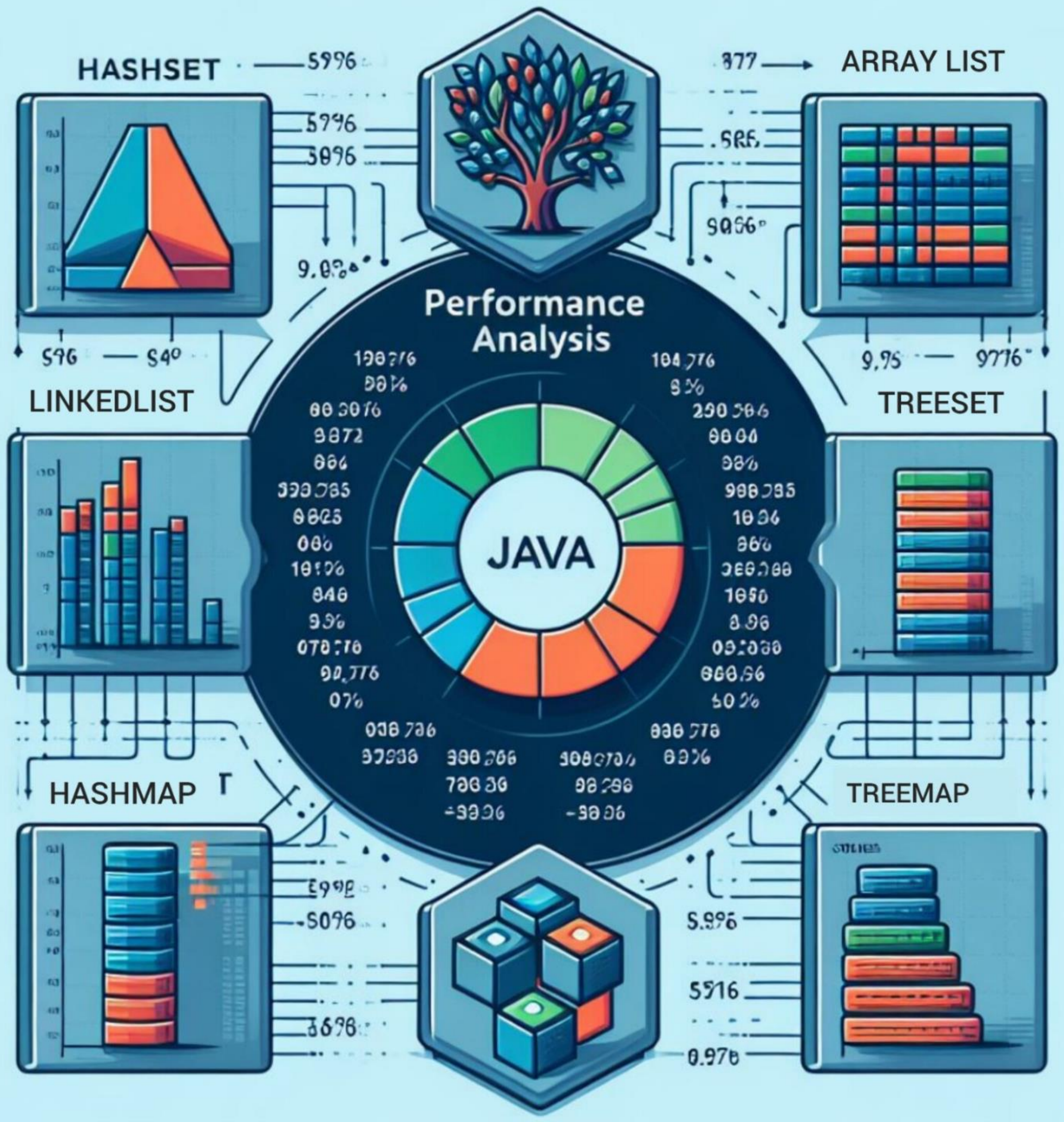
In22-S2-CS1040

Project 02

Performance Analysis of Java Collections



Project Team: Code Busters

220271C: JAYASUNDARA J.W.K.B.R

220230C: HIMAN E.A.A

220606K: SHAMILA N.A.B

220315R: KAVINDA L.G.N

Contents

1.0 Description of program design.......... 4
2.0 Full Java program code used for testing 5
2.1 Hashset 5
2.2 Treeset 8
2.3 Linked Hash set.......... 11
2.4 ArrayList 14
2.5 Linked List 17
2.6 ArrayDeque 20
2.7 PriorityQueue 23
2.8 Hashmap.......... 26
2.9 Treemap.......... 29
2.10 LinkedHashMap 32
3.0 Comparison Table of Performance Data.......... 35
3.1 Comparison Chart – Java Collection-wise 35
3.2 Comparison Chart – Implementation-wise 36
4.0 Discussions on the reasons for performance variations.......... 37
4.1 Add Element.......... 37
5.0 Remove Element.......... 39
5.1 Contain Elements 40
5.2 Clear all elements in Array.......... 42
5.3 Concise summary 43
5.3.1 Remove an element from array.......... 44
6.0 References.......... 45

Table of Figures

Figure 3-1.......... 35
Figure 3-2.......... 36
Figure 4-1.......... 37
Figure 4-2.......... 39
Figure 4-3.......... 40
Figure 4-4.......... 42

Table of Tables

Table 3-1 *Times are shown in nanoseconds. 35
Table 5-1 - Average time complexity. 43
Table 5-2 - Average time complexity 44

1.0 Description of program design

This project compares various Java collections to see which ones perform better for tasks like adding items, checking for items, removing items, and clearing out all items. The collections we're testing include

1. HashSet
2. TreeSet
3. LinkedHashSet
4. ArrayList
5. LinkedList
6. ArrayDeque
7. PriorityQueue
8. HashMap
9. TreeMap
10. LinkedHashMap

To test the performance of Java Collection implementations, we create a program that follows these steps:

1. **Initialization:** We create 100,000 random Integer objects and store them in an array. Then, we create instances of each Collection implementation.

2. **Add Test:** For each collection, we iterate through the array of Integer objects and measure the time it takes to add each element. we repeat this process 100 times and calculate the average time.

3. **Contains Test:** We randomly select an Integer object from the array and check if the collection contains it. We repeat this process 100 times and calculate the average time.

4. **Remove Test:** Similarly, we randomly select an Integer object from the array and remove it from the collection if it exists. We repeat this process 100 times and calculate the average time.

5. **Clear Test:** We measure the time it takes to clear all elements from each collection. We repeat this process 100 times and calculate the average time.

6. **Output Results:** Finally, We display the average time taken for each test for every Collection implementation using graph and chart.

2.0 Full Java program code used for testing

2.1 Hashset

```java
import java.util.HashSet;
import java.util.Random;

public class hashSetPerformance {
    long startTime, endTime, totalTime;

    public static void addTime(HashSet<Integer> hashSet) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            hashSet.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            hashSet.remove(element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");

    }

    public static void removeTime(HashSet<Integer> hashSet) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            hashSet.remove(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            hashSet.add(element);
        }
        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");

    }

    public static void containsTime(HashSet<Integer> hashSet) {

        Random random = new Random();


        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            hashSet.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to contain element: " +
averageTime + " nanoseconds");

    }

    public static void clearTime(HashSet<Integer> hashSet) {

        Random random = new Random();


        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            hashSet.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (hashSet.size() < 100000) {
                hashSet.add(random.nextInt(100000));
            }
        }
        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear element: " + averageTime
+ " nanoseconds");

    }


    public static void main(String[] args) {
        Random random = new Random();
        HashSet<Integer> hashSet1 = new HashSet<>();

        while (hashSet1.size() < 100000) {
            hashSet1.add(random.nextInt(100000));
        }

        addTime(hashSet1);
        removeTime(hashSet1);
        containsTime(hashSet1);
        clearTime(hashSet1);
    }
}
```

output

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac hashSetPerformance.java &&
java hashSetPerformance
Average time taken to add element: 133 nanoseconds
Average time taken to remove element: 470 nanoseconds
Average time taken to contain element: 400 nanoseconds
Average time taken to clear element: 94358 nanoseconds
```

2.2 Treeset

```
import java.util.Random;
import java.util.TreeSet;

public class treeSetPerformance {
    public static void addTime(TreeSet<Integer> treeSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            treeSet.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            treeSet.remove(element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(TreeSet<Integer> treeSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            treeSet.remove(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            treeSet.add(element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(TreeSet<Integer> treeSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            treeSet.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(TreeSet<Integer> treeSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            treeSet.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (treeSet.size() < 100000) {
                treeSet.add(random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        TreeSet<Integer> treeSet1 = new TreeSet<>();

        while (treeSet1.size() < 100000) {
            treeSet1.add(random.nextInt(100000));
        }

        addTime(treeSet1);
        removeTime(treeSet1);
        containsTime(treeSet1);
        clearTime(treeSet1);
    }
}
```

```
Output
```

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac treeSetPerformance.java &&
java treeSetPerformance
Average time taken to add element: 97 nanoseconds
Average time taken to remove element: 1281 nanoseconds
Average time taken to check contains: 565 nanoseconds
Average time taken to clear elements: 2325 nanoseconds
```

2.3 Linked Hash set

```
import java.util.LinkedHashSet;
import java.util.Random;

public class LinkedHashSetPerformance {
    public static void addTime(LinkedHashSet<Integer> linkedHashSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            linkedHashSet.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedHashSet.remove(element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(LinkedHashSet<Integer> linkedHashSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            linkedHashSet.remove(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedHashSet.add(element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(LinkedHashSet<Integer> linkedHashSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            linkedHashSet.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(LinkedHashSet<Integer> linkedHashSet) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            linkedHashSet.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (linkedHashSet.size() < 100000) {
                linkedHashSet.add(random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        LinkedHashSet<Integer> linkedHashSet1 = new LinkedHashSet<>();

        while (linkedHashSet1.size() < 100000) {
            linkedHashSet1.add(random.nextInt(100000));
        }

        addTime(linkedHashSet1);
        removeTime(linkedHashSet1);
        containsTime(linkedHashSet1);
        clearTime(linkedHashSet1);
    }
}
```

```
output
```

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac LinkedHashSetPerformance.java
&& java LinkedHashSetPerformance
Average time taken to add element: 192 nanoseconds
Average time taken to remove element: 467 nanoseconds
Average time taken to check contains: 281 nanoseconds
Average time taken to clear elements: 95536 nanoseconds
```

2.4 ArrayList

```
import java.util.ArrayList;
import java.util.Random;

public class ArrayListPerformance {
    public static void addTime(ArrayList<Integer> arrayList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            arrayList.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            arrayList.remove(Integer.valueOf(element));
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(ArrayList<Integer> arrayList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int index = random.nextInt(arrayList.size());
            int element = arrayList.get(index);
            long startTime = System.nanoTime();
            arrayList.remove(index);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            arrayList.add(index, element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(ArrayList<Integer> arrayList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            arrayList.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(ArrayList<Integer> arrayList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            arrayList.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (arrayList.size() < 100000) {
                arrayList.add(random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        ArrayList<Integer> arrayList1 = new ArrayList<>();

        while (arrayList1.size() < 100000) {
            arrayList1.add(random.nextInt(100000));
        }

        addTime(arrayList1);
        removeTime(arrayList1);
        containsTime(arrayList1);
        clearTime(arrayList1);
    }
}
```

Output

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac ArrayListPerformance.java &&
java ArrayListPerformance
Average time taken to add element: 96 nanoseconds
Average time taken to remove element: 16506 nanoseconds
Average time taken to check contains: 82357 nanoseconds
Average time taken to clear elements: 33578 nanoseconds
```

2.5 Linked List

```java
import java.util.LinkedList;
import java.util.Random;

public class LinkedListPerformance {
    public static void addTime(LinkedList<Integer> linkedList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            linkedList.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedList.remove(Integer.valueOf(element));
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(LinkedList<Integer> linkedList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int index = random.nextInt(linkedList.size());
            int element = linkedList.get(index);
            long startTime = System.nanoTime();
            linkedList.remove(index);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedList.add(index, element);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(LinkedList<Integer> linkedList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            linkedList.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(LinkedList<Integer> linkedList) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            linkedList.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (linkedList.size() < 100000) {
                linkedList.add(random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        LinkedList<Integer> linkedList1 = new LinkedList<>();

        while (linkedList1.size() < 100000) {
            linkedList1.add(random.nextInt(100000));
        }

        addTime(linkedList1);
        removeTime(linkedList1);
        containsTime(linkedList1);
        clearTime(linkedList1);
    }
}
```

```
Output
```

```
Running] cd "c:\Users\HP\Downloads\Code\" && javac LinkedListPerformance.java &&
java LinkedListPerformance
Average time taken to add element: 41 nanoseconds
Average time taken to remove element: 49450 nanoseconds
Average time taken to check contains: 151188 nanoseconds
Average time taken to clear elements: 222290 nanoseconds
```

2.6 ArrayDeque

```
import java.util.ArrayDeque;
import java.util.Random;

public class ArrayDequePerformance {
    public static void addTime(ArrayDeque<Integer> arrayDeque) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            arrayDeque.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            arrayDeque.remove(Integer.valueOf(element)); // Remove by value to
maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(ArrayDeque<Integer> arrayDeque) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            arrayDeque.remove(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            arrayDeque.add(element); // Add back to maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(ArrayDeque<Integer> arrayDeque) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            arrayDeque.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(ArrayDeque<Integer> arrayDeque) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            arrayDeque.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (arrayDeque.size() < 100000) {
                 arrayDeque.add(random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        ArrayDeque<Integer> arrayDeque1 = new ArrayDeque<>();

        while (arrayDeque1.size() < 100000) {
            arrayDeque1.add(random.nextInt(100000));
        }

        addTime(arrayDeque1);
        removeTime(arrayDeque1);
```

```java
        containsTime(arrayDeque1);
        clearTime(arrayDeque1);
    }
}
```

Output

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac ArrayDequePerformance.java &&
java ArrayDequePerformance
Average time taken to add element: 57 nanoseconds
Average time taken to remove element: 28862 nanoseconds
Average time taken to check contains: 80445 nanoseconds
Average time taken to clear elements: 52742 nanoseconds
```

2.7 PriorityQueue

```java
import java.util.PriorityQueue;
import java.util.Random;

public class PriorityQueuePerformance {
    public static void addTime(PriorityQueue<Integer> priorityQueue) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            priorityQueue.add(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            priorityQueue.remove(element); // Remove to maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(PriorityQueue<Integer> priorityQueue) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            priorityQueue.remove(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            priorityQueue.add(element); // Add back to maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(PriorityQueue<Integer> priorityQueue) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int element = random.nextInt(100000);
            long startTime = System.nanoTime();
            priorityQueue.contains(element);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check contains: " + averageTime
+ " nanoseconds");
    }

    public static void clearTime(PriorityQueue<Integer> priorityQueue) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            priorityQueue.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (priorityQueue.size() < 100000) {
                priorityQueue.add(random.nextInt(100000));
            }
        }
        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");

    }

    public static void main(String[] args) {
        Random random = new Random();
        PriorityQueue<Integer> priorityQueue1 = new PriorityQueue<>();

        while (priorityQueue1.size() < 100000) {
            priorityQueue1.add(random.nextInt(100000));
        }

        addTime(priorityQueue1);
        removeTime(priorityQueue1);
        containsTime(priorityQueue1);
        clearTime(priorityQueue1);
    }
}
```

```
Output
```

```
Running] cd "c:\Users\HP\Downloads\Code\" && javac PriorityQueuePerformance.java
&& java PriorityQueuePerformance
Average time taken to add element: 48 nanoseconds
Average time taken to remove element: 73169 nanoseconds
Average time taken to check contains: 42110 nanoseconds
Average time taken to clear elements: 33815 nanoseconds
```

2.8 Hashmap

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class HashMapPerformance {
    public static void putTime(HashMap<Integer, Integer> hashMap) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(10000) + 100000;
            int value = random.nextInt(100000);
            long startTime = System.nanoTime();
            hashMap.put(key, value);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            hashMap.remove(key); // Remove to maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to put entry: " + averageTime + "
nanoseconds");
    }

    public static void removeTime(HashMap<Integer, Integer> hashMap) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            int value = hashMap.get(key);
            long startTime = System.nanoTime();
            hashMap.remove(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            hashMap.put(key, value); // Add back to maintain consistent size
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove entry: " + averageTime +
" nanoseconds");
    }
    public static void getTime(HashMap<Integer, Integer> hashMap) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            long startTime = System.nanoTime();
            hashMap.get(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to get entry: " + averageTime + "
nanoseconds");
    }

    public static void clearTime(HashMap<Integer, Integer> hashMap) {
        Random random = new Random();
        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            hashMap.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (hashMap.size() < 100000) {
                int key = random.nextInt(100000);
                int value = random.nextInt(100000);
                hashMap.put(key, value);
            }
        }
        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        HashMap<Integer, Integer> hashMap1 = new HashMap<>();

        while (hashMap1.size() < 100000) {
            int key = random.nextInt(100000);
            int value = random.nextInt(100000);
            hashMap1.put(key, value);
        }

        putTime(hashMap1);
        removeTime(hashMap1);
        getTime(hashMap1);
        clearTime(hashMap1);
    }
}
```

Output

```
[Running] cd "c:\Users\HP\Downloads\Code\Over\" && javac HashMapPerformance.java
&& java HashMapPerformance
Average time taken to put entry: 187 nanoseconds
Average time taken to remove entry: 262 nanoseconds
Average time taken to get entry: 175 nanoseconds
Average time taken to clear elements: 71343 nanoseconds
```

2.9 Treemap

```
import java.util.Random;
import java.util.TreeMap;

public class TreeMapPerformance {

    public static void addTime(TreeMap<Integer, Integer> treeMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(10000) + 100000;
            int value = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            treeMap.put(key, value);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            treeMap.remove(key);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(TreeMap<Integer, Integer> treeMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            long startTime = System.nanoTime();
            treeMap.remove(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            treeMap.put(key, random.nextInt(100000)); // Re-add to keep the size
consistent
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(TreeMap<Integer, Integer> treeMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            long startTime = System.nanoTime();
            treeMap.containsKey(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check containment: " +
averageTime + " nanoseconds");
    }

    public static void clearTime(TreeMap<Integer, Integer> treeMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            treeMap.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (treeMap.size() < 100000) {
                int key = random.nextInt(100000);
                treeMap.put(key, random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
```

```java
        TreeMap<Integer, Integer> treeMap = new TreeMap<>();

        // Adding elements to TreeMap
        while (treeMap.size() < 100000) {
            int key = random.nextInt(100000);
            treeMap.put(key, random.nextInt(100000));
        }

        // Benchmarking operations
        addTime(treeMap);
        removeTime(treeMap);
        containsTime(treeMap);
        clearTime(treeMap);
    }
}
```

Output

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac TreeMapPerformance.java &&
java TreeMapPerformance
Average time taken to add element: 134 nanoseconds
Average time taken to remove element: 1084 nanoseconds
Average time taken to check containment: 418 nanoseconds
Average time taken to clear elements: 55 nanoseconds
```

2.10 LinkedHashMap

```
import java.util.LinkedHashMap;
import java.util.Random;

public class LinkedHashMapPerformance {

    public static void addTime(LinkedHashMap<Integer, Integer> linkedHashMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(10000) + 100000;
            int value = random.nextInt(10000) + 100000;
            long startTime = System.nanoTime();
            linkedHashMap.put(key, value);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedHashMap.remove(key);
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to add element: " + averageTime +
" nanoseconds");
    }

    public static void removeTime(LinkedHashMap<Integer, Integer> linkedHashMap)
{
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            long startTime = System.nanoTime();
            linkedHashMap.remove(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            linkedHashMap.put(key, random.nextInt(100000)); // Re-add to keep the
size consistent
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to remove element: " + averageTime
+ " nanoseconds");
    }

    public static void containsTime(LinkedHashMap<Integer, Integer>
linkedHashMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            int key = random.nextInt(100000);
            long startTime = System.nanoTime();
            linkedHashMap.containsKey(key);
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to check containment: " +
averageTime + " nanoseconds");
    }

    public static void clearTime(LinkedHashMap<Integer, Integer> linkedHashMap) {
        Random random = new Random();

        long totalTime = 0;

        for (int i = 0; i < 100; i++) {
            long startTime = System.nanoTime();
            linkedHashMap.clear();
            long endTime = System.nanoTime();
            totalTime += endTime - startTime;
            while (linkedHashMap.size() < 100000) {
                int key = random.nextInt(100000);
                linkedHashMap.put(key, random.nextInt(100000));
            }
        }

        long averageTime = totalTime / 100;
        System.out.println("Average time taken to clear elements: " + averageTime
+ " nanoseconds");
    }

    public static void main(String[] args) {
        Random random = new Random();
        LinkedHashMap<Integer, Integer> linkedHashMap = new LinkedHashMap<>();

        // Adding elements to LinkedHashMap
        while (linkedHashMap.size() < 100000) {
            int key = random.nextInt(100000);
            linkedHashMap.put(key, random.nextInt(100000));
        }

        // Benchmarking operations
        addTime(linkedHashMap);
        removeTime(linkedHashMap);
        containsTime(linkedHashMap);
        clearTime(linkedHashMap);
    }
}
```

Output

```
[Running] cd "c:\Users\HP\Downloads\Code\" && javac LinkedHashMapPerformance.java
&& java LinkedHashMapPerformance
Average time taken to add element: 170 nanoseconds
Average time taken to remove element: 385 nanoseconds
Average time taken to check containment: 258 nanoseconds
Average time taken to clear elements: 93043 nanoseconds
```

3.0 Comparison Table of Performance Data

Java Collection	*Add element*	*remove element*	*contains element*	*clear element*
Hashset	133	470	400	94358
TreeSet	97	1281	565	2325
LinkedHashSet	192	467	281	95536
ArrayList	96	16506	82357	33578
LinkedList	41	49450	151188	222290
ArrayDeque	57	28862	80445	52742
PriorityQueue	48	73169	42110	33815
HashMap	187	262	175	71343
TreeMap	134	1084	418	55
LinkedHashMap	170	385	258	93043

*Table 3-1 *Times are shown in nanoseconds.*

3.1 Comparison Chart – Java Collection-wise

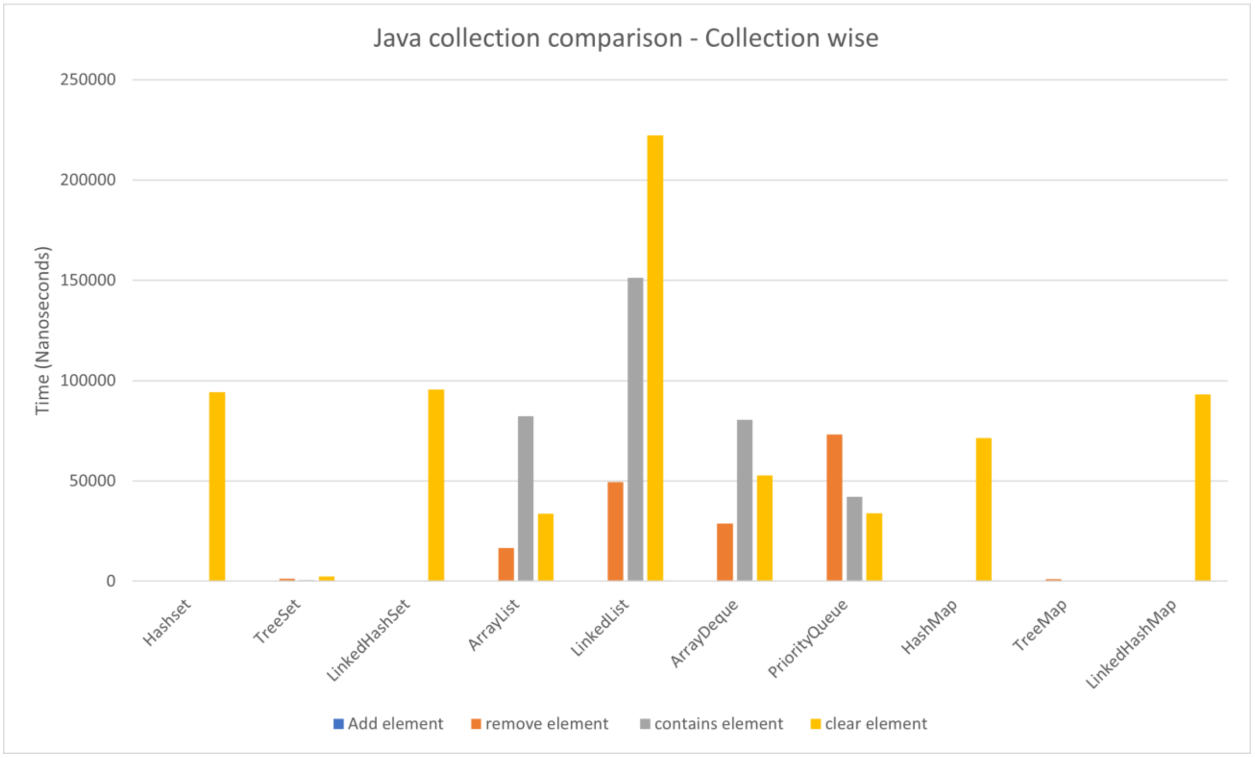


Figure 3-1

3.2 <u>Comparison Chart – Implementation-wise</u>

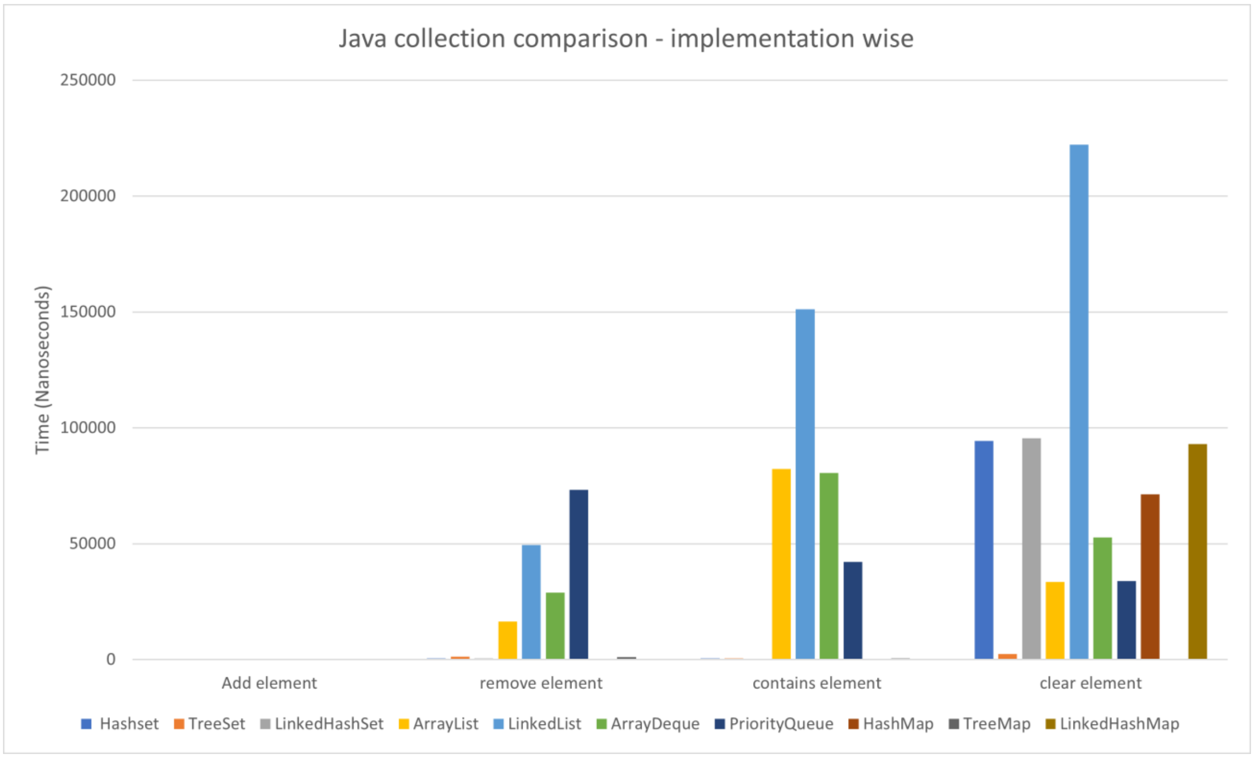
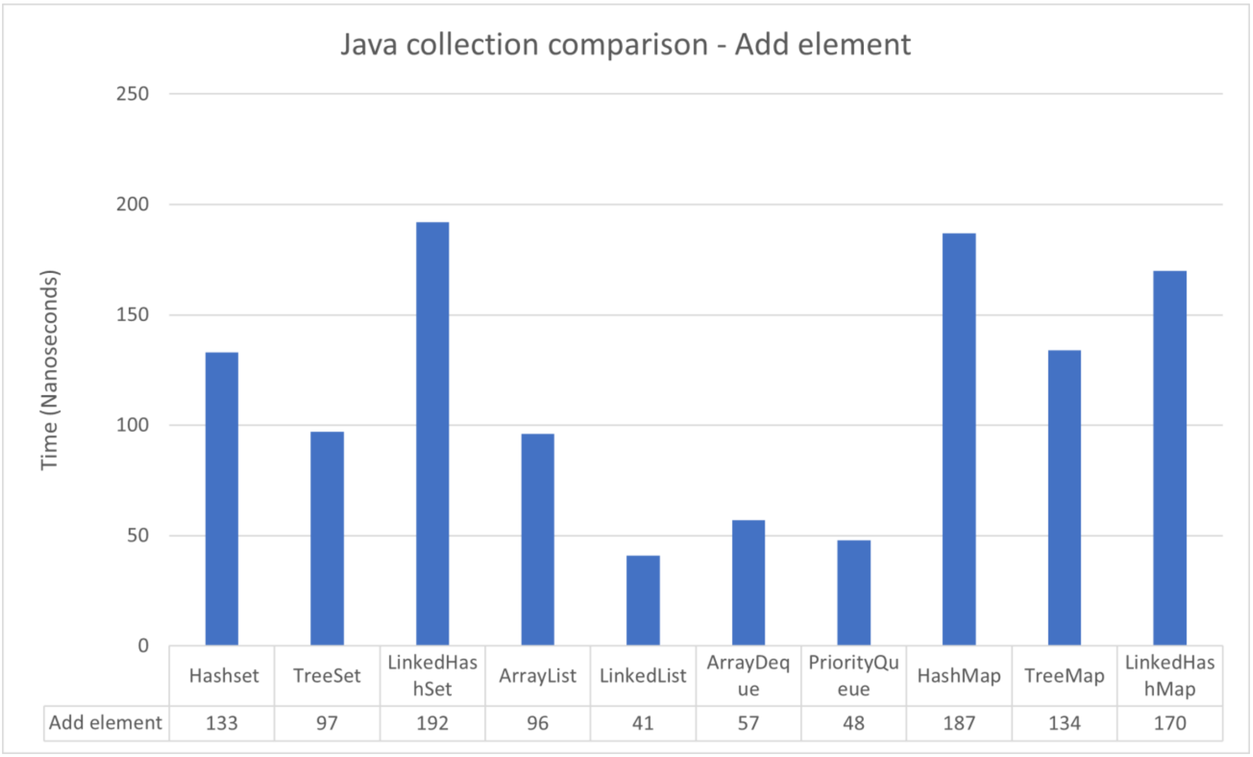


Figure 3-2

4.0 Discussions on the reasons for performance variations

4.1 Add Element



	Hashset	TreeSet	LinkedHashSet	ArrayList	LinkedList	ArrayDeque	PriorityQueue	HashMap	TreeMap	LinkedHashMap
Add element	133	97	192	96	41	57	48	187	134	170

Figure 4-1

```
LinkedList < PriorityQueue < ArrayDeque < ArrayList < TreeSet < Hashset < TreeMap
< LinkedHashMap < HashMap < LinkedHashSet
```

LinkedList is the fastest at adding elements because it's designed to efficiently add items at the end. **LinkedHashSet**, on the other hand, is **slower** because it needs **to maintain uniqueness while adding elements**, which requires more work.

HashMap, **LinkedHashMap**, and **HashSet** also have **slower performance** because they store elements in **key-value pairs** instead of just individual elements. This means there's extra overhead in managing these pairs, making the process slower compared to simpler data structures like LinkedList.

Here, we'll examine why each Java implementation has its own performance characteristics.

LinkedList (41 ns): The fastest in this list because it uses a linked list data structure which allows constant time insertion at any position of the list.

PriorityQueue (48 ns): Faster than ArrayDeque because it uses a heap data structure which allows logarithmic time insertion while maintaining the order of elements based on their priorities.

ArrayDeque (57 ns): Faster than ArrayList because it uses a circular array which avoids shifting elements when adding at the front or back of the deque.

ArrayList (96 ns): Faster than most of the set implementations because it uses an array data structure which allows constant time access and insertion at the end of the list.

TreeSet (97 ns): Faster than HashSet because it uses a balanced binary search tree which has a logarithmic time complexity for add operations.

Hashset (133 ns): This collection is relatively slow because it uses hashing mechanism which involves calculating hash codes and resolving collisions for each element.

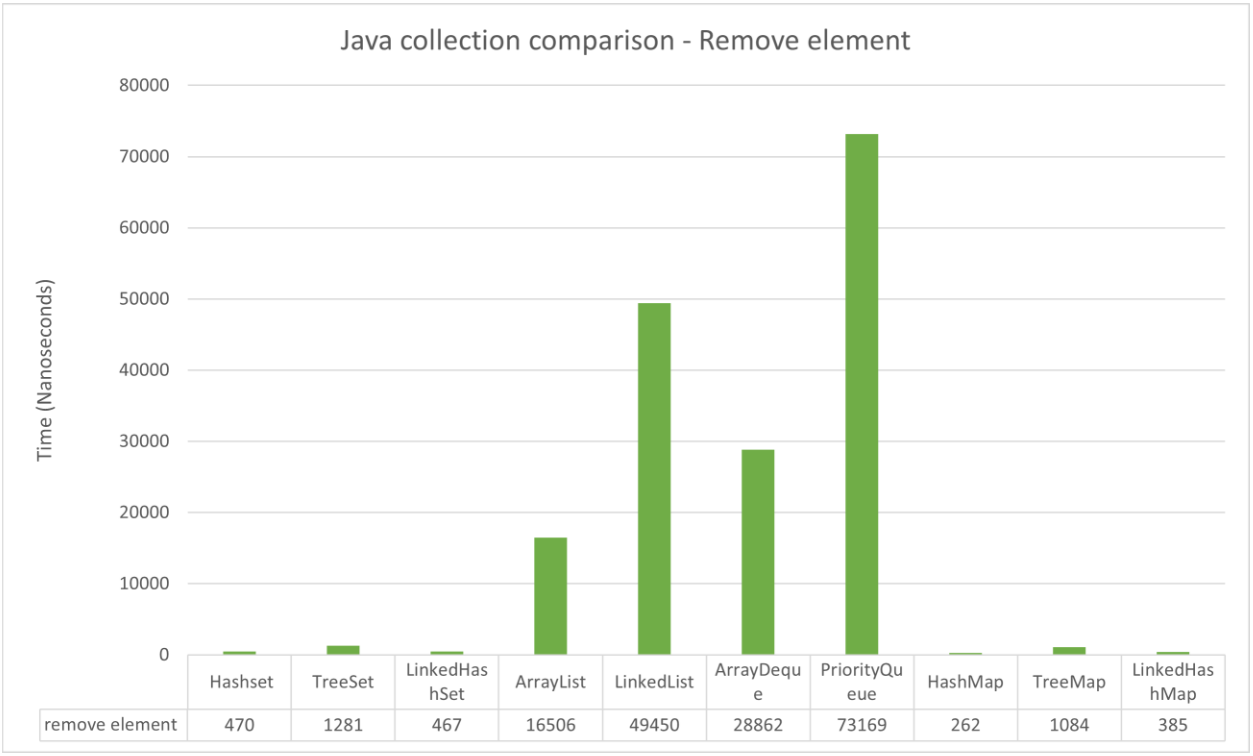
TreeMap (134 ns): Similar to TreeSet but faster because it uses a red-black tree which is a self-balancing binary search tree that reduces the height of the tree and improves the performance of add operations.

LinkedHashMap (170 ns): Similar to LinkedHashSet but slower because it handles key-value pairs instead of single elements.

HashMap (187 ns): Similar to HashSet but slower because it handles key-value pairs instead of single elements.

LinkedHashSet (192 ns): Slower than both HashSet and TreeSet because it maintains a doubly-linked list along with the hash table, which adds more overhead for each element.

5.0 Remove Element



	Hashset	TreeSet	LinkedHashSet	ArrayList	LinkedList	ArrayDeque	PriorityQueue	HashMap	TreeMap	LinkedHashMap
remove element	470	1281	467	16506	49450	28862	73169	262	1084	385

Figure 5-1

```
HashMap < LinkedHashMap < LinkedHashSet < Hashset < TreeMap < TreeSet < ArrayList
< ArrayDeque < LinkedList < PriorityQueue
```

When it comes to removing elements in Java implementations, the observed performance varies across different data structures. This ordering reflects how efficiently each data structure handles the removal of elements.

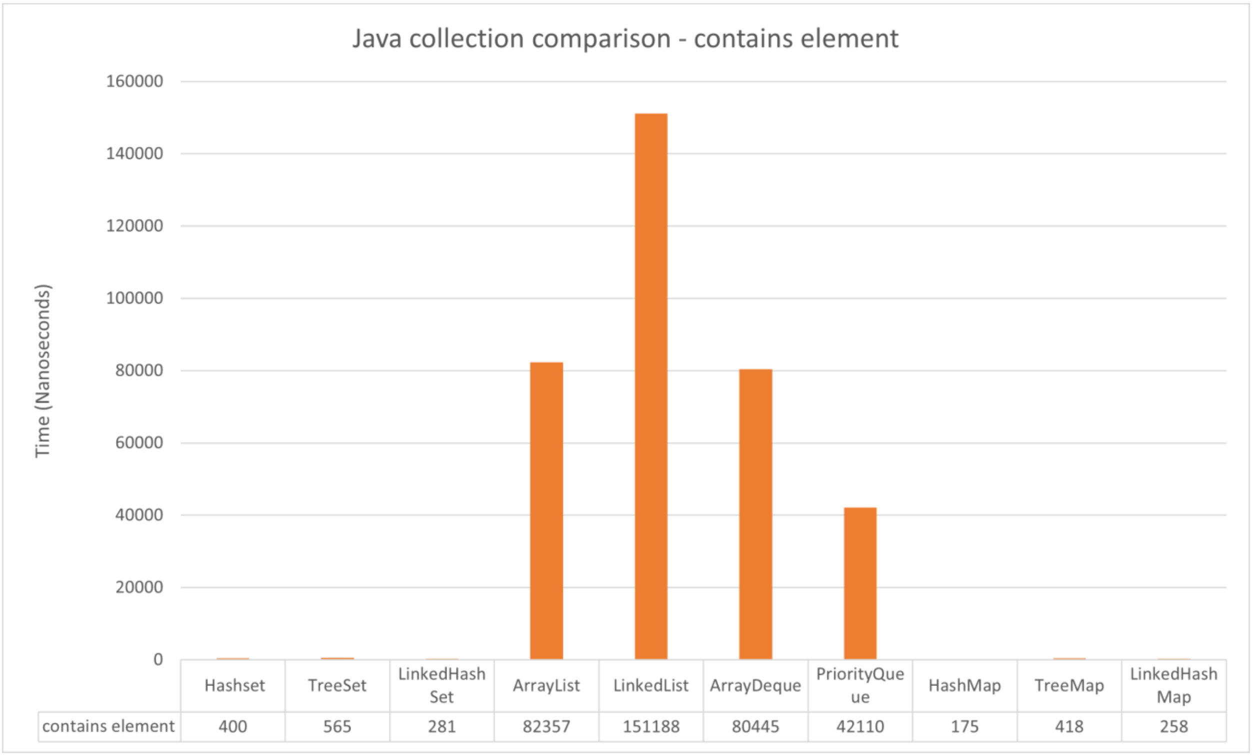
For instance, **HashMap** and **LinkedHashMap** excel because they use hashing to quickly locate and remove elements **based on their keys.**

LinkedHashSet and **HashSet** also perform relatively well due to their use of hashing for element removal, albeit with additional considerations for maintaining uniqueness.

TreeMap and **TreeSet** exhibit slightly lower performance as they **rely on tree structures**, which involve more complex operations for removal.

Finally, **ArrayList, ArrayDeque, LinkedList,** and **PriorityQueue** exhibit slower removal times due to the need for shifting elements or maintaining specific orders, which can be more time-consuming operations.

5.1 Contain Elements



	Hashset	TreeSet	LinkedHash Set	ArrayList	LinkedList	ArrayDeque	PriorityQue ue	HashMap	TreeMap	LinkedHash Map
contains element	400	565	281	82357	151188	80445	42110	175	418	258

Figure 5-2

```
HashMap < LinkedHashMap < LinkedHashSet < Hashset < TreeMap < TreeSet <
PriorityQueue < ArrayDeque < ArrayList < LinkedList
```

When checking for the presence of elements in an array within Java implementations, the observed performance varies across different data structures.

HashMap and **LinkedHashMap** use **method hashing**, which makes it quick to check if something's there based on a key.

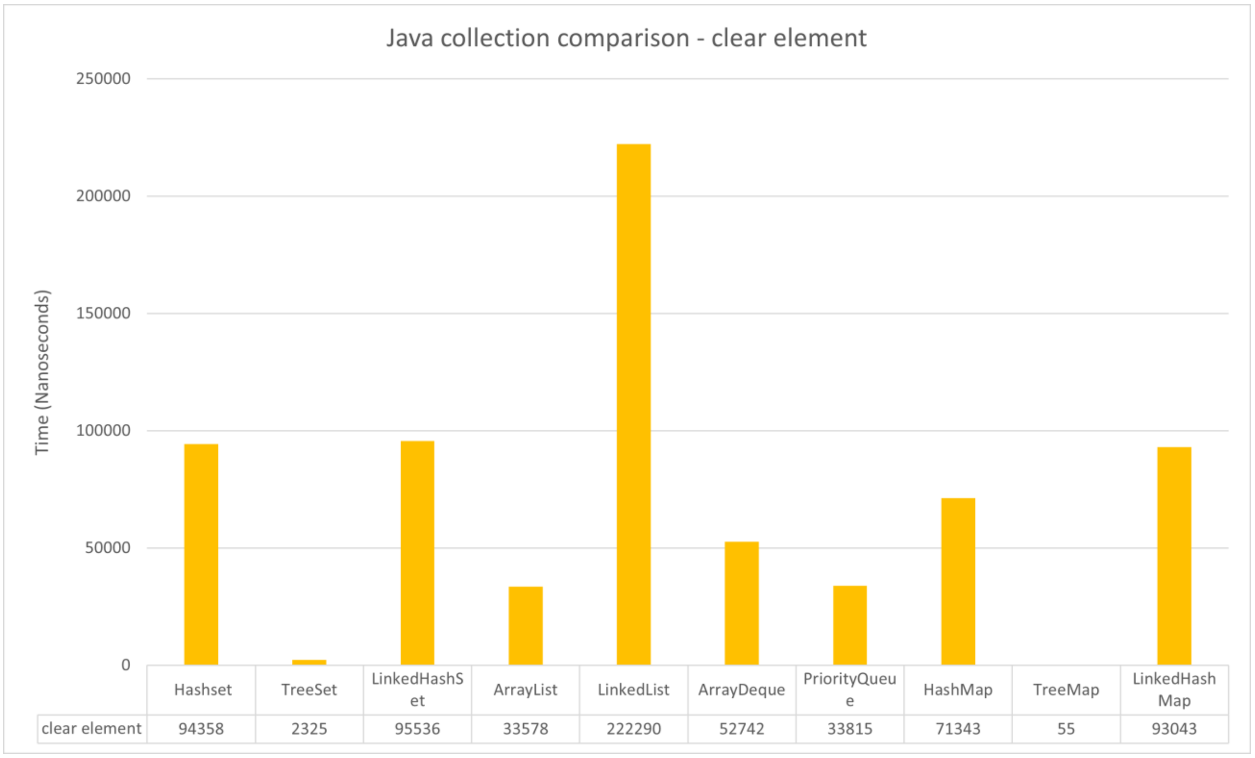
LinkedHashSet and **HashSet** are also fast because they use similar methods, but they also need to make sure each element is unique.

TreeMap and **TreeSet** demonstrate slightly lower performance **because they use trees**, which need more steps to find things.

PriorityQueue, **ArrayDeque**, **ArrayList**, and **LinkedList** exhibit slower containment times due to the need for sequential scanning or maintaining specific orders, which can be more time-intensive operations.

Overall, **HashMap**, **LinkedHashMap**, **LinkedHashSet**, **HashSet**, **TreeMap**, and **TreeSet** all have similar ways of checking if an element is in the array, using hashing or tree-based methods for quick searches.

5.2 Clear all elements in Array



	Hashset	TreeSet	LinkedHashSet	ArrayList	LinkedList	ArrayDeque	PriorityQueue	HashMap	TreeMap	LinkedHashMap
clear element	94358	2325	95536	33578	222290	52742	33815	71343	55	93043

Figure 5-3

```
TreeMap < TreeSet < ArrayList < PriorityQueue < ArrayDeque < HashMap <
LinkedHashMap < Hashset < LinkedHashSet < LinkedList
```

When it comes to clearing all elements in an array using Java, the time taken varies across different data structures.

TreeMap and **TreeSet** perform well due to their **tree structures**, which allow for fast clearing operations.

ArrayList and **PriorityQueue** also clear quickly since they don't need to deal with complex data structures. **ArrayDeque** follows suit with efficient clearing times. **HashMap** and **LinkedHashMap** perform decently, **utilizing hashing** for swift removal of elements.

HashSet and **LinkedHashSet** clear somewhat slower due to their need for **uniqueness maintenance.**

Lastly, **LinkedList** takes the most time to clear as it requires **traversing through each element one by one.**

5.3 Concise summary

	Implementation	Add elements	Contain elements	Clear all elements
List	**ArrayList**	O(1)	O(n)	O(n)
	LinkedList	O(1)	O(n)	O(n)
Set	**Hashset**	O(1)	O(1)	O(n)
	LinkedHashSet	O(1)	O(1)	O(n)
	TreeSet	O(log n)	O(log n)	O(n)
Map	**HashMap**	O(1)	O(1)	O(n)
	LinkedHashMap	O(1)	O(1)	O(n)
	TreeMap	O(log n)	O(log n)	O(n)
Queue	**PriorityQueue**	O(log n)	O(log n)	O(n)
Stack or Queue	**ArrayDeque**	O(1)	O(n)	O(n)

Table 5-1 - Average time complexity.

5.3.1 Remove an element from array

Implementation	**Remove by value**	**Remove by index**	**Remove by iterator**
HashSet	O(1)	N/A	O(1)
HashMap	O(1)	N/A	O(1)
LinkedHashSet	O(1)	N/A	O(1)
LinkedHashMap	O(1)	N/A	O(1)
TreeSet	O(log n)	N/A	O(1)
TreeMap	O(log n)	N/A	O(1)
PriorityQueue	O(log n)	N/A	O(n)
ArrayList	O(n)	O(n)	O(1)
LinkedList	O(n)	O(n)	O(1)
ArrayDeque	O(n)	N/A	O(1)

Table 5-2 - Average time complexity

6.0 References

Aibin, M. (2024, 01 08). *Choosing the Right Java Collection*. Retrieved from baeldung: https://www.baeldung.com/java-choose-list-set-queue-map

big O notation for removing an element from a linked list [duplicate]. (n.d.). Retrieved from stackoverflow: https://stackoverflow.com/questions/40465415/big-o-notation-for-removing-an-element-from-a-linked-list

Big-O summery for java Collections Framework implementations? (n.d.). Retrieved from stackoverflow: https://stackoverflow.com/questions/559839/big-o-summary-for-java-collections-framework-implementations

SortedSet remove() method in Java with Examples. (2021, 01 21). Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/sortedset-remove-method-in-java-with-examples/