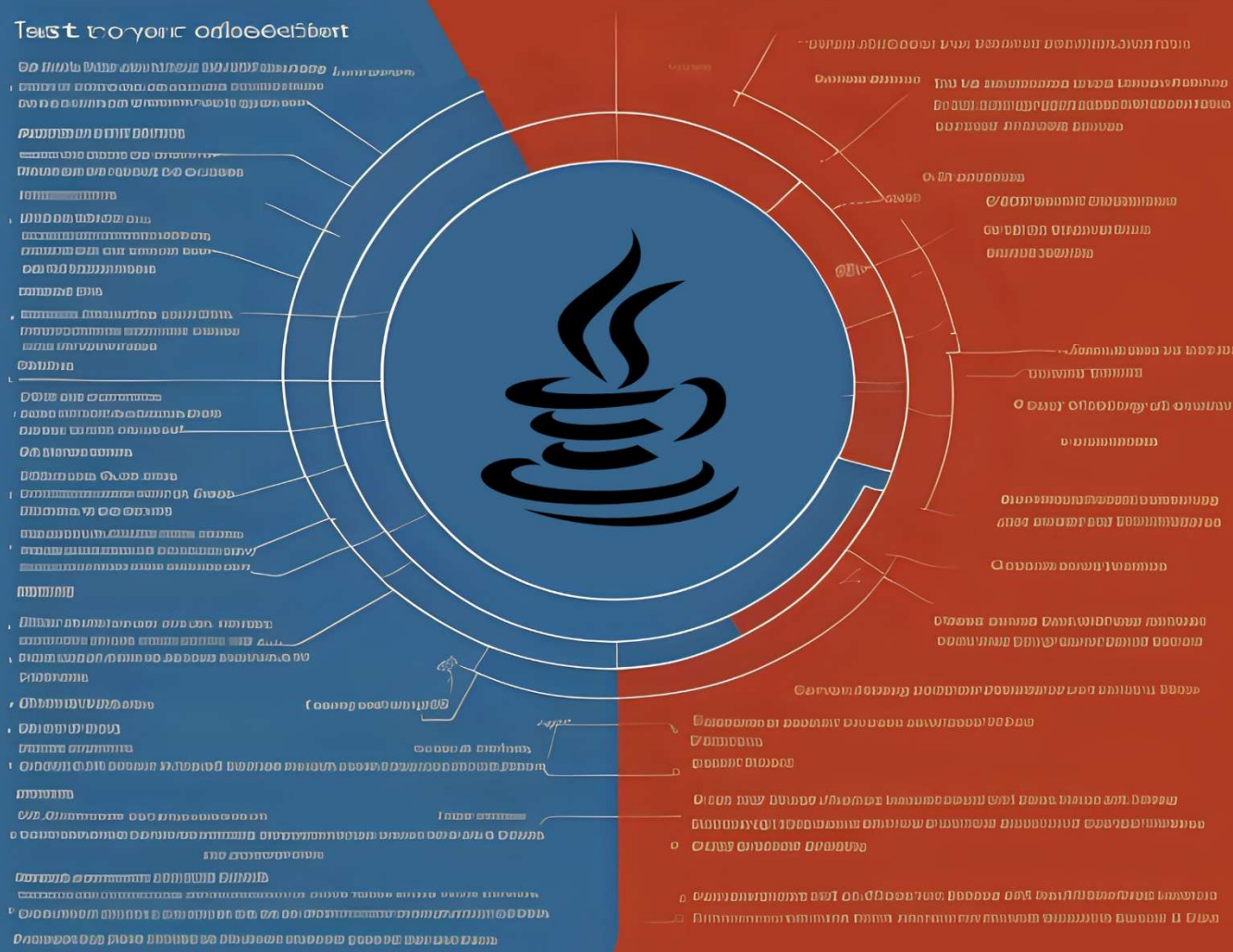
Benchmarking Java Collections



SCORPION

Project : Benchmarking Java Collections
Team : Scorpion
Contribution : Jayasinghe J.P.T.S. - 220262B
Edirisinghe E.K.K.D.R - 220152M
De Silva W.D.T. - 220104U
Heshan M.A.D.V. - 220224L

01. Program Design

The test was conducted using Java. The program has ten different files; HashSetUtil, TreeSetUtil, LinkedHashSetUtil, ArrayListUtil, LinkedListUtil, ArrayDequeUtil, PriorityQueueUtil, HashMapUtil, TreeMapUtil, LinkedHashMapUtil.

Each sub-program is designed to evaluate the runtime performance of various methods (add, contain, remove, and clear) on the selected collections (HashSet, TreeSet, LinkedHashSet, ArrayList, LinkedList, ArrayDeque, PriorityQueue, HashMap, TreeMap, LinkedHashMap) in Java. The primary objective is to measure and compare the time taken for these operations to execute on these collections across multiple iterations. A mean run time was calculated to compare the collections with each other.

The key aspects of the program can be broken down as below.

- A method to create an object of the relevant collection containing random integers ranging up to 100,000.
- Evolution Methods for add, contain, remove, and clear methods.
    - Specific operation is executed on the collection for 100 iterations
    - The run time for each iteration is recorded in an array
    - Average run time for each operation is calculated
- The above methods are called inside the 'main' method and the results are displayed in a formatted way
- Each collection was loaded with 100,000 items in the test. The time measurements were obtained at nanoseconds. The tests were performed using default initial capacity and load factor values.

Classes were created for each collection to obtain the runtime for operations.

Then a test class was created to implement the specified methods for each collection to compare the runtimes.

02. Program Used for Testing

➢ HashSetUtil

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

public class HashSetUtil {
    public static HashSet<Integer> createHashSet(int length){
        HashSet<Integer> random_hash_set = new HashSet<Integer>();
        for(int j=0;j<length;++j){
            Random random = new Random();
            random_hash_set.add(random.nextInt(100000));
        }
        return random_hash_set;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashSet<Integer> generatedRandomHashSet = createHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashSet", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashSet<Integer> generatedRandomHashSet = createHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;
```

```
        System.out.printf("%-15s%-20s%-10s\n", "HashSet", "Contain", meanRunTime);
    }

    public static void evalRemove(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashSet<Integer> generatedRandomHashSet = createHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashSet", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashSet<Integer> generatedRandomHashSet = createHashSet(100000);
            long start_time = System.nanoTime();
            generatedRandomHashSet.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashSet", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

➢ TreeSetUtil

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

public class TreeSetUtil {
    public static TreeSet<Integer> createTreeSet(int length){
        TreeSet<Integer> random_tree_set = new TreeSet<>();
        for(int j=0;j<length;++j){
            Random random = new Random();
            random_tree_set.add(random.nextInt(100000));
        }
        return random_tree_set;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeSet<Integer> generatedRandomHashSet = createTreeSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeSet", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeSet<Integer> generatedRandomHashSet = createTreeSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeSet", "Contain", meanRunTime);
    }

    public static void evalRemove(){
```

```
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeSet<Integer> generatedRandomHashSet = createTreeSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeSet", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeSet<Integer> generatedRandomHashSet = createTreeSet(100000);
            long start_time = System.nanoTime();
            generatedRandomHashSet.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeSet", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **LinkedHashSetUtil**

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

public class LinkedHashSetUtil {
    public static LinkedHashSet<Integer> createLinkedHashSet(int length) {
        LinkedHashSet<Integer> randomLinkedHashSet = new LinkedHashSet<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomLinkedHashSet.add(random.nextInt(100000));
        }
        return randomLinkedHashSet;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashSet<Integer> generatedRandomHashSet =
createLinkedHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashSet", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashSet<Integer> generatedRandomHashSet =
createLinkedHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashSet", "Contain",
meanRunTime);
```

```
    }

    public static void evalRemove(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashSet<Integer> generatedRandomHashSet =
createLinkedHashSet(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashSet.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashSet", "Remove",
meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashSet<Integer> generatedRandomHashSet = createLinkedHashSet(100000);
            long start_time = System.nanoTime();
            generatedRandomHashSet.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashSet", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **ArrayListUtil**

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

public class ArrayListUtil {
    public static ArrayList<Integer> createArrayList(int length) {
        ArrayList<Integer> randomArrayList = new ArrayList<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomArrayList.add(random.nextInt(100000));
        }
        return randomArrayList;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayList<Integer> generatedRandomArrayList = createArrayList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayList.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayList", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayList<Integer> generatedRandomArrayList = createArrayList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayList.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayList", "Contain", meanRunTime);
    }

    public static void evalRemove(){
```

```
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayList<Integer> generatedRandomArrayList = createArrayList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayList.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayList", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayList<Integer> generatedRandomArrayList = createArrayList(100000);
            long start_time = System.nanoTime();
            generatedRandomArrayList.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayList", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **LinkedListUtil**

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

public class LinkedListUtil {
    public static LinkedList<Integer> createLinkedList(int length) {
        LinkedList<Integer> randomLinkedList = new LinkedList<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomLinkedList.add(random.nextInt(100000));
        }
        return randomLinkedList;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedList<Integer> generatedRandomLinkedList = createLinkedList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomLinkedList.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedList", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedList<Integer> generatedRandomLinkedList = createLinkedList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomLinkedList.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedList", "Contain", meanRunTime);
    }

    public static void evalRemove(){
```

```
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedList<Integer> generatedRandomLinkedList = createLinkedList(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomLinkedList.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedList", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedList<Integer> generatedRandomLinkedList = createLinkedList(100000);
            long start_time = System.nanoTime();
            generatedRandomLinkedList.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedList", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **ArrayDequeList**

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

public class ArrayDequeUtil {
    public static ArrayDeque<Integer> createArrayDeque(int length) {
        ArrayDeque<Integer> randomArrayDeque = new ArrayDeque<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomArrayDeque.add(random.nextInt(100000));
        }
        return randomArrayDeque;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayDeque<Integer> generatedRandomArrayDeque = createArrayDeque(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayDeque.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayDeque", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayDeque<Integer> generatedRandomArrayDeque = createArrayDeque(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayDeque.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayDeque", "Contain", meanRunTime);
    }

    public static void evalRemove(){
```

```
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayDeque<Integer> generatedRandomArrayDeque = createArrayDeque(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomArrayDeque.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayDeque", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            ArrayDeque<Integer> generatedRandomArrayDeque = createArrayDeque(100000);
            long start_time = System.nanoTime();
            generatedRandomArrayDeque.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "ArrayDeque", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **PriorityQueueUtil**

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

public class PriorityQueueUtil {
    public static PriorityQueue<Integer> createPriorityQueue(int length) {
        PriorityQueue<Integer> randomPriorityQueue = new PriorityQueue<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomPriorityQueue.add(random.nextInt(100000));
        }
        return randomPriorityQueue;
    }
    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            PriorityQueue<Integer> generatedRandomPriorityQueue =
createPriorityQueue(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomPriorityQueue.add(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "PriorityQueue", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            PriorityQueue<Integer> generatedRandomPriorityQueue =
createPriorityQueue(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomPriorityQueue.contains(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "PriorityQueue", "Contain",
meanRunTime);
    }
```

```
    public static void evalRemove(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            PriorityQueue<Integer> generatedRandomPriorityQueue =
createPriorityQueue(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomPriorityQueue.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "PriorityQueue", "Remove",
meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            PriorityQueue<Integer> generatedRandomPriorityQueue =
createPriorityQueue(100000);
            long start_time = System.nanoTime();
            generatedRandomPriorityQueue.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "PriorityQueue", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **HashMapUtil**

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

public class HashMapUtil {
    public static HashMap<Integer, Integer> createHashMap(int length) {
        HashMap<Integer, Integer> randomHashMap = new HashMap<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomHashMap.put(j, random.nextInt(100000));
        }
        return randomHashMap;
    }
    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashMap<Integer, Integer> generatedRandomHashMap = createHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashMap.put(random.nextInt(100000),random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashMap", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashMap<Integer, Integer> generatedRandomHashMap = createHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashMap.containsKey(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashMap", "Contain", meanRunTime);
    }

    public static void evalRemove(){
        long[] run_times = new long[100];
```

```
        for (int i = 0; i < 100; ++i) {
            HashMap<Integer, Integer> generatedRandomHashMap = createHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomHashMap.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashMap", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            HashMap<Integer, Integer> generatedRandomHashMap = createHashMap(100000);
            long start_time = System.nanoTime();
            generatedRandomHashMap.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "HashMap", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **TreeMapUtil**

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

public class TreeMapUtil {
    public static TreeMap<Integer, Integer> createTreeMap(int length) {
        TreeMap<Integer, Integer> randomTreeMap = new TreeMap<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomTreeMap.put(j, random.nextInt(100000));
        }
        return randomTreeMap;
    }

    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeMap<Integer,Integer> generatedRandomTreeMap = createTreeMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomTreeMap.put(100001,random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeMap", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeMap<Integer,Integer> generatedRandomTreeMap = createTreeMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomTreeMap.containsKey(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeMap", "Contain", meanRunTime);
    }

    public static void evalRemove(){
```

```
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeMap<Integer,Integer> generatedRandomTreeMap = createTreeMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomTreeMap.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeMap", "Remove", meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            TreeMap<Integer,Integer> generatedRandomTreeMap = createTreeMap(100000);
            long start_time = System.nanoTime();
            generatedRandomTreeMap.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "TreeMap", "Clear", meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
        }
    }
```

- **LinkedHashMapUtil**

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

public class LinkedHashMapUtil {
    public static LinkedHashMap<Integer, Integer> createLinkedHashMap(int length) {
        LinkedHashMap<Integer, Integer> randomLinkedHashMap = new LinkedHashMap<>();
        Random random = new Random();
        for (int j = 0; j < length; ++j) {
            randomLinkedHashMap.put(j, random.nextInt(100000));
        }
        return randomLinkedHashMap;
    }
    public static long avgTime(long[] array){
        long totalRuntime = 0;
        for (long i : array){
            totalRuntime += i;
        }
        return totalRuntime;
    }

    public static void evalAdd(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashMap<Integer,Integer> generatedRandomLinkedHashMap =
createLinkedHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();

generatedRandomLinkedHashMap.put(random.nextInt(100000),random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashedMap", "Add", meanRunTime);
    }


    public static void evalContain(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashMap<Integer,Integer> generatedRandomLinkedHashMap =
createLinkedHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomLinkedHashMap.containsValue(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashedMap", "Contain",
meanRunTime);
```

```
    }

    public static void evalRemove(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashMap<Integer,Integer> generatedRandomLinkedHashMap =
createLinkedHashMap(100000);
            long start_time = System.nanoTime();
            Random random = new Random();
            generatedRandomLinkedHashMap.remove(random.nextInt(100000));
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;

        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashedMap", "Remove",
meanRunTime);
    }

    public static void evalClear(){
        long[] run_times = new long[100];
        for (int i = 0; i < 100; ++i) {
            LinkedHashMap<Integer,Integer> generatedRandomLinkedHashMap =
createLinkedHashMap(100000);
            long start_time = System.nanoTime();
            generatedRandomLinkedHashMap.clear();
            long end_time = System.nanoTime();
            long run_time = end_time - start_time;
            run_times[i] = run_time;
        }

        long RunTime = avgTime(run_times);

        long meanRunTime = RunTime / run_times.length;
        System.out.printf("%-15s%-20s%-10s\n", "LinkedHashedMap", "Clear",
meanRunTime);
    }

    public static void main(String a[]){
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        evalAdd();
        evalContain();
        evalRemove();
        evalClear();
        System.out.print(ANSI_RESET);
    }
}
```

- **Test**

```
public class Test {
    public static void main(String[] args) {
        //add
        String ANSI_PENCIL = "\u001b[38;2;253;182;0m";
        String ANSI_RED = "\u001b[38;5;147m";
        String ANSI_RESET = "\u001B[0m";
        String ANSI_GREEN = "\u001B[32m";
        System.out.println(ANSI_GREEN);
        System.out.println("*****************  "+"Add"+"  *****************");
        System.out.println(ANSI_RESET);

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        HashSetUtil.evalAdd();
        TreeSetUtil.evalAdd();
        LinkedHashSetUtil.evalAdd();
        ArrayListUtil.evalAdd();
        LinkedListUtil.evalAdd();
        ArrayDequeUtil.evalAdd();
        PriorityQueueUtil.evalAdd();
        HashMapUtil.evalAdd();
        TreeMapUtil.evalAdd();
        LinkedHashedMapUtil.evalAdd();
        System.out.print(ANSI_RESET);


        //contain
        System.out.println(ANSI_GREEN);
        System.out.println("*****************  "+"Contain"+"  *****************");
        System.out.println(ANSI_RESET);

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        HashSetUtil.evalContain();
        TreeSetUtil.evalContain();
        LinkedHashSetUtil.evalContain();
        ArrayListUtil.evalContain();
        LinkedListUtil.evalContain();
        ArrayDequeUtil.evalContain();
        PriorityQueueUtil.evalContain();
        HashMapUtil.evalContain();
        TreeMapUtil.evalContain();
        LinkedHashedMapUtil.evalContain();
        System.out.print(ANSI_RESET);


        //remove
        System.out.println(ANSI_GREEN);
        System.out.println("*****************  "+"Remove"+"  *****************");
        System.out.println(ANSI_RESET);

        System.out.print(ANSI_PENCIL);
        System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
        System.out.print(ANSI_RED);
        HashSetUtil.evalRemove();
        TreeSetUtil.evalRemove();
```

```
        LinkedHashSetUtil.evalRemove();
        ArrayListUtil.evalRemove();
        LinkedListUtil.evalRemove();
        ArrayDequeUtil.evalRemove();
        PriorityQueueUtil.evalRemove();
        HashMapUtil.evalRemove();
        TreeMapUtil.evalRemove();
        LinkedHashedMapUtil.evalRemove();
        System.out.print(ANSI_RESET);


         //clear
         System.out.println(ANSI_GREEN);
         System.out.println("*****************  "+"Clear"+"  *****************");
         System.out.println(ANSI_RESET);

         System.out.print(ANSI_PENCIL);
         System.out.printf("%-15s%-20s%-10s\n", "Collection", "Method", "Mean Run Time
(ns)");
         System.out.print(ANSI_RED);
         HashSetUtil.evalClear();
         TreeSetUtil.evalClear();
         LinkedHashSetUtil.evalClear();
         ArrayListUtil.evalClear();
         LinkedListUtil.evalClear();
         ArrayDequeUtil.evalClear();
         PriorityQueueUtil.evalClear();
         HashMapUtil.evalClear();
         TreeMapUtil.evalClear();
         LinkedHashedMapUtil.evalClear();
         System.out.print(ANSI_RESET);
    }
}
```

03. Behaviour of Operations in Different Collections

Addition

1. **HashSet:**
    - A hash function is used internally by the HashSet **add()** method to identify which bucket the element should be placed in. It verifies unique elements and looks for duplication.
2. **TreeSet:**
    - An internal Red-Black Tree is used by the TreeSet **add()** method. A custom comparator or the elements' natural order is used to determine the sorting order of the elements.
3. **LinkedHashSet:**
    - The LinkedHashSet **add()** method relies on a hash function, just like the HashSet add() method, but it additionally keeps a doubly-linked list to maintain the insertion order.
4. **ArrayList:**
    - **add()** method in ArrayList appends the element to the dynamic array. If the array is full, it should be resized. Then a new array is created and the elements are copied to the new array.
5. **LinkedList:**
    - **add()** method in LinkedList adds a new node to the end of the linked list. It will update the references to maintain the structure of the linked list.
6. **ArrayDeque:**
    - **add()** method in ArrayDeque adds the element to the end of the deque. If required, the internal array is resized.
7. **PriorityQueue:**
    - **add()** method in PriorityQueue inserts the element based on its priority. After each insertion, heap property is maintained.
8. **HashMap:**
    - **put()** method in HashMap insert data after calculating the hash of the key to determine the bucket. If there's a hash collision, it resolves it using different techniques (separate chaining or open addressing). If necessary, it may need to resize the internal array.

9. **TreeMap:**
    - **put()** method in TreeMap adds the key-value pair to the Red-Black Tree. A sorted order is maintained based on the keys.
10. **LinkedHashMap:**
    - **put()** method in LinkedHashMap is similar to that of HashMap. In addition to calculating the hash of the key, it also maintains a doubly linked list to preserve the order of insertion.

Contain

1. **HashSet:**
    - **contains()** method in HashSet uses the hash code of the element to check the corresponding bucket. Then equality checks are performed to see if the element is present.
2. **TreeSet:**
    - **contains()** method in TreeSet uses a self-balancing binary search, typically a Red-Black Tree to find the element based on its natural order or a custom comparator.
3. **LinkedHashSet:**
    - Similar to the **contains()** method in HashSet, LinkedHashSet also uses the hash code to locate the specific bucket. Then it iterates through the linked list while checking for equality.
4. **ArrayList:**
    - **contains()** method in ArrayList iterates through the elements linearly. It performs equality checks until the element is found or the end of the list is reached.
5. **LinkedList:**
    - **contains()** method in LinkedList traverses through the linked list starting from the head, until the element is found or the end of the list is reached.
6. **ArrayDeque:**
    - **contains()** method in ArrayDeque iterates over the internal array while performing equality checks until the element is found or the end of the deque is reached.

7. **PriorityQueue:**
    - **contains()** method in PriorityQueue iterates through the heap structure to find the element based on the priority.
8. **HashMap:**
    - **containsKey()** method in HashMap calculates the hash of the key, determines the bucket, and then checks for equality within the entries of that bucket.
9. **TreeMap:**
    - **containsKey()** method in TreeMap checks whether a key is present using a binary search algorithm based on the natural order or a custom comparator.
10. **LinkedHashMap:**
    - **containsKey()** method in LinkedHashMap uses the hash code to locate the bucket and then iterates through the linked list in that bucket.

Remove

1. **HashSet:**
    - **remove()** method in HashSet uses the hash code of the element to locate the corresponding bucket. It then iterates through the elements in that bucket and removes the relevant element.
2. **TreeSet:**
    - **remove()** method in TreeSet uses a binary search algorithm to find and remove the element based on its natural order or a custom comparator.
3. **LinkedHashSet:**
    - Similar to HashSet, **remove()** in LinkedHashSet calculates the hash key and locates the relevant bucket. Then it iterates through the linked list in the bucket and removes the element based on equality.
4. **ArrayList:**
    - **remove()** method in ArrayList uses linear search to find the index of the element and then removes it. After removing, the subsequent elements are shifted to fill the gap.

5. **LinkedList:**
    - LinkedList's **remove()** method traverses through the linked list to find the node that contains the element. Then it adjusts the references to remove the node.
6. **ArrayDeque:**
    - **remove()** method in ArrayDeque removes the element from the internal array and shifts the elements to fill the gap if necessary.
7. **PriorityQueue:**
    - **remove()** method in PriorityQueue removes and returns the element with the highest priority. The priority within the queue is then maintained by reorganizing the heap.
8. **HashMap:**
    - **remove()** method in HashMap uses the hash code of the key to locate the bucket. It then iterates through the elements in the bucket and removes the relevant element.
9. **TreeMap:**
    - **remove()** method in TreeMap uses a binary search algorithm to find and remove the element based on the natural order or a custom comparator.
10. **LinkedHashMap:**
    - **remove()** method in LinkedHashMap uses the hash code to locate the bucket and then iterates through the linked list in the bucket to remove the relevant element.

Clear

1. **HashSet:**
    - **clear()** method in HashSet involves resetting the internal array of buckets to an initial state. Each bucket is set to null.
2. **TreeSet:**
    - **clear()** method in TreeSet resets the root of the tree to null, effectively removing all elements.
3. **LinkedHashSet:**

- **clear()** method in LinkedHashSet resets the head and tail of the linked list to null, while maintaining the linked structure for preserving insertion order.

4. **ArrayList:**

    - **clear()** method in ArrayList sets the size of the dynamic array to zero, effectively removing all elements. Any references to objects might still exist in memory, but the list itself becomes empty.

5. **LinkedList:**

    - **clear()** method in LinkedList removes all elements individually and updates the linked structure while iterating through the list.

6. **ArrayDeque:**

    - **clear()** method in ArrayDeque sets the size of the internal array to zero, effectively removing all elements.

7. **PriorityQueue:**

    - **clear()** method in PriorityQueue removes all elements from the internal heap structure. Removing elements from a priority queue effectively adjusts the heap structure to maintain the heap properties (min-heap or max-heap), ensuring that the remaining elements maintain the priority order.

8. **HashMap:**

    - **clear()** method in HashMap sets the size to zero, effectively removing all key-value pairs.

9. **TreeMap:**

    - **clear()** method in TreeMap removes all elements by resetting the size of the internal structure to zero.

10. **LinkedHashMap:**

    - **clear()** method in LinkedHashMap involves resetting the head and tail of the linked list to null, and clearing the internal hash table and the linked structure.

04. Time complexity of operations

Method / Collection	Add		Contain		Remove		Clear
	Avg case	Worst case	Avg case	Worst case	Avg case	Worst case	Avg case and Worst case
HashSet	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
LinkedHashSet	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)
ArrayList	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)
LinkedList	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(1)
ArrayDeque	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(1)
PriorityQueue	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)	O(n)
HashMap	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
LinkedHashMap	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	O(log n)

05. Run time for different operations

Method / Collection	Add	Contain	Remove	Clear
HashSet	7396	15719	17921	54512
TreeSet	6698	16042	18117	6899
LinkedHashSet	5084	10009	11807	88775
ArrayList	2350	218327	25936	67038
LinkedList	1833	51411	116195	679310
ArrayDeque	2442	177235	195954	85386
PriorityQueue	3144	247645	223378	93573
HashMap	4681	8519	13880	174410
TreeMap	2389	144989	4459	4814
LinkedHashMap	5406	76082	1960	4790

Note: All the measurements are in nanoseconds

06. Analyzing Performance Variations

Here is the order of the runtimes for the Add method.

LinkedList < ArrayList < TreeMap < ArrayDeque < PriorityQueue < HashMap < LinkedHashSet < LinkedHashMap < TreeSet < HashSet

- ArrayList and LinkedLists use the least runtime because the element is appended to the end of the array.
- TreeMap, PriorityQueue, and TreeSet require more time since elements need to be rearranged after add method.
- Hash collisions may have caused Hashmaps, LinkedHashSets, and HashSets to take the longest runtime.

The runtime of the contain method will be as follows.

HashMap < LinkedHashSet < HashSet < TreeSet < LinkedList < LinkedHashMap < TreeMap < ArrayDeque < ArrayList < PriorityQueue

- HashMap, LinkedHashSet, and HashSet have the least run time due to the hash-based indexing mechanism.
- As TreeSet and TreeMap are based on binary search trees (Red-Black-Tree) take a little longer runtime.
- ArrayDeque, ArrayList, and LinkedList have taken more time since it implements the linear search
- PriorityQueue takes the longest runtime as it had optimized only for retrieving and removing the minimum element efficiently. Therefore, it searches by retrieving the smallest element. The runtime will be longer for this.

The Remove method's run time will be as follows.

LinkedHashMap < TreeMap < LinkedHashSet < HashMap < HashSet < TreeSet < ArrayList < LinkedList < ArrayDeque < PriorityQueue

- Because of the hash-based indexing approach, LinkedHashMap, TreeMap, LinkedHashSet, HashMap, and HashSet use the least amount of runtime to locate and remove elements.
- TreeSet and TreeMap are a little bit slow due to binary tree implementation. So, after removing the element, the collection should be rearranged to maintain the sorted structure.
- In ArrayList, LinkedList, and PriorityQueue, the required element is found using linear search. After removing the element, the internal structure is reorganized. Therefore, their runtimes are much higher than the others.

The Clear method's run time will be as follows.

LinkedHashMap < TreeMap < TreeSet < HashSet < ArrayList < ArrayDeque < LinkedHashSet < PriorityQueue < HashMap < LinkedList

- LinkedHashMap, TreeMap, and TreeSet rely on balanced tree structures that allow efficient element removal in O(log n) time. This explains their likely superior performance in clearing elements compared to collections with linear structures.
- HashSet uses key lookups for removal, making it faster than collections iterating through all elements for smaller data sizes
- ArrayList and ArrayDeque both rely on internal arrays. Clearing involves setting the size to zero. Although the time complexity for these two collections is O(1), factors like array resizing and memory management can impact the performance of the operation
- LinkedHashSet is similar to HashSet, but it maintains element insertion order with additional linked structure. Clearing might involve iterating through this linked structure, affecting the performance.
- In PriorityQueue, clearing involves manipulating the internal heap structure as mentioned in section 03.
- HashMap uses hashing for lookups but it might involve additional steps to remove entries from the underlying data structure.
- Due to the linked structure of LinkedLists, clearing requires iterating through each element and removing it individually, making it the slowest option in most cases.

07. Conclusion

This project successfully evaluated the runtime performance of various methods (add, contain, remove, and clear) on different Java collections.

The observations are based on the specific configurations and data sizes used in the test. Also note that factors like specific implementations, optimizations, garbage collection behaviour, and the hardware of the device running the program (CPU, memory, etc.) can influence the performance of above operations.

References:

GeeksforGeeks. Collections in Java. https://www.geeksforgeeks.org/collections-in-java-2/

psayre23, github. Runtime Complexity of Java Collections. https://gist.github.com/psayre23/c30a821239f4818b0709

Yogesh Kumar, Medium. (2020, Nov 16). Time and Space Complexity of Collections. https://yogeshkkhichi.medium.com/time-and-space-complexity-of-collections-5a00c7b1d32b

ChatGPT. (2024, Feb 14). Clear Method Complexity Summary. https://chat.openai.com/share/40cb5efc-1124-4e69-920d-becbd0f47a4d