TEAM
404-ERROR

17, FEBRARY, 2024

PROJECT #2

JAVA COLLECTIONS COMPARISON

Group members :

- 220135N- Dissanayake D.M.A.K.
- 220168R - FERNANDO N.P.A.
- 220522A - RATHNAYAKA R.M.P.B.
- 220525K - RATHNAYAKE R.M.D.D.

CONTENT

1. Program design

The program is designed to systematically evaluate the performance of different Java Collection implementations across four key methods: ‘add’, ‘contains’, ‘remove’, and ‘clear’. It follows a structured approach:

1. **Initialization:** Load each collection with 100,000 random integers before testing.
2. **Testing Phase:**
    - Execute each method 100 times for accurate time measurements.
    - Record the time taken for each operation using System.nanoTime().
3. **Data Recording:**
    - Aggregate the execution times for each method and collection.
    - Store the data in a structured format, associating each result with the corresponding collection and method.
4. **Reporting:**
    - Compile the recorded data into a report with sections dedicated to each method.
    - Present the average execution times for each collection under test.
    - Offer insights into performance disparities among different collection implementations.
5. **Error Handling:** Implement mechanisms to handle potential errors gracefully during testing.
6. **Modularity and Documentation:**
    - Design modular code for reusability and maintainability.
    - Provide inline comments and clear documentation for ease of understanding and future reference.

This design ensures thorough testing and analysis of Java Collection implementations, enabling informed decisions regarding their suitability for specific use cases.

2. Java program code

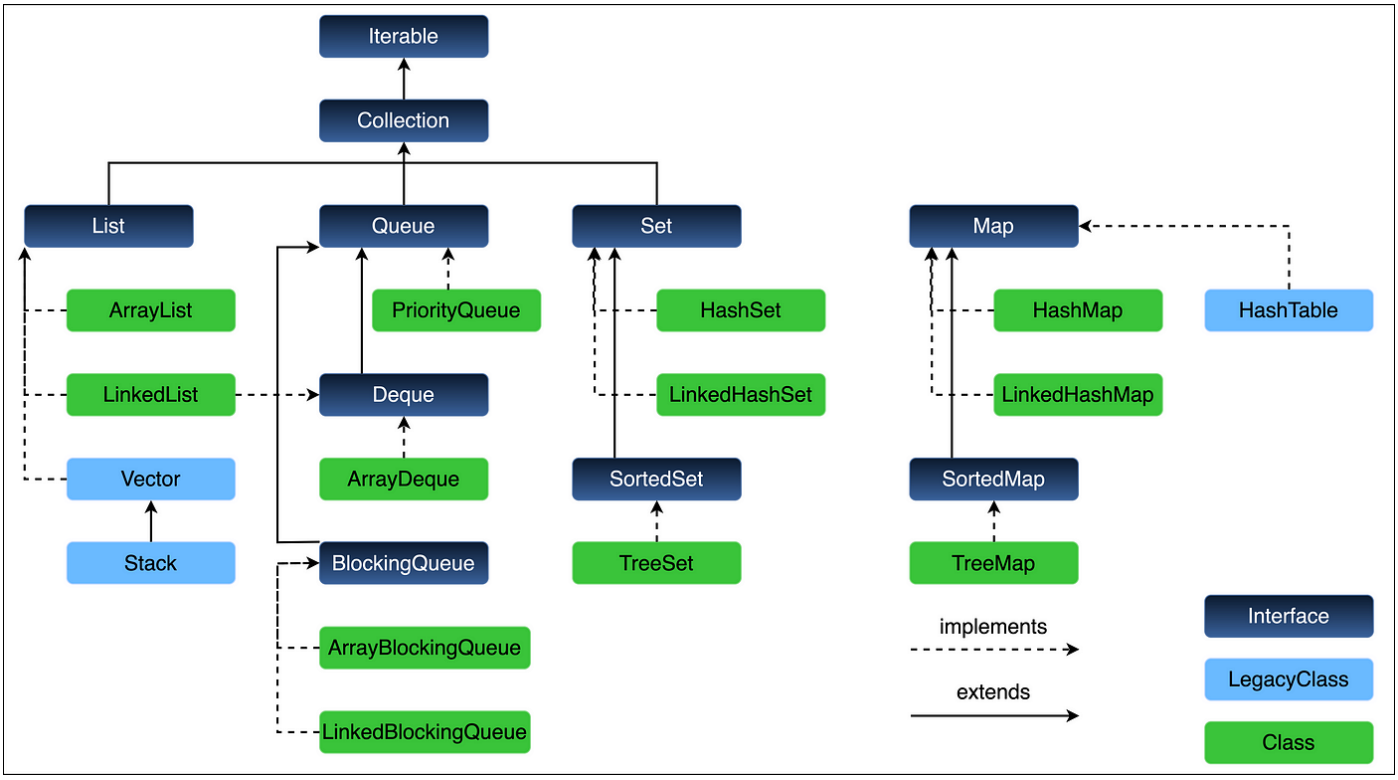
```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

public class Main {

    public static void main(String[] args) {
        long[][] avgTimes = new long[10][4];
        for(int t = 0; t < 100; t++){
            int numberOfItems = 100;

            // Create a list of Java Collection implementations
            List<Collection<Integer>> collections = new ArrayList<>();
            collections.add(new HashSet<>());
            collections.add(new TreeSet<>());
            collections.add(new LinkedHashSet<>());
            collections.add(new ArrayList<>());
            collections.add(new LinkedList<>());
            collections.add(new ArrayDeque<>());
            collections.add(new PriorityQueue<>());

            // Load each collection with 100,000 random Integer objects

            for (int i = 0; i < numberOfItems; i++) {
                int value = (int) (Math.random() * numberOfItems);
                for (Collection<Integer> collection : collections) {
                    collection.add(value);
                }

            }
            for (Collection<Integer> collection : collections) {
                System.out.println(collection);
            }
            List<Integer> array = new ArrayList<>(collections.get(3));

            // Test the map implementations
            List<Map<Integer, Integer>> maps = new ArrayList<>();
            maps.add(new HashMap<>());
            maps.add(new TreeMap<>());
            maps.add(new LinkedHashMap<>());
//Load each map with 100,000 random key-value pairs

for (Map<Integer, Integer> map : maps) {
    int value = (int) (Math.random() * numberOfItems);
    for (int i = 0; i < numberOfItems; i++) {
        map.put((int) (i+1),value);
    }
}
```

```
            // Perform the tests for each method and print the time taken
            for (int i = 0; i < collections.size(); i++) {
                avgTimes[i][0] += (testAdd(collections.get(i)));
                avgTimes[i][1] += (testContains(collections.get(i)));
                avgTimes[i][2] += (testRemove(collections.get(i)));
                avgTimes[i][3] += (testClear(collections.get(i)));
            }

            // Perform the tests for each method and print the time taken
            for (int i = 7; i < maps.size()+7; i++) {
                avgTimes[i][0] += (testPut(maps.get(i-7)));
                avgTimes[i][1] += (testContainsKey(maps.get(i-7)));
                avgTimes[i][2] += (testRemoveMap(maps.get(i-7)));
                avgTimes[i][3] += (testClearMap(maps.get(i-7)));
            }
        }
        for(long[] arr : avgTimes){
            for(long l : arr){
                System.out.printf("%d ,", l/100);
            }
            System.out.println();
        }
        graphPlot(avgTimes);
        System.out.println("Graph.txt has been created");
    }

    // Define the methods for collections.
    public static long testAdd(Collection<Integer> collection) {
        long startTime = System.nanoTime();
        collection.add((int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testContains(Collection<Integer> collection) {
        long startTime = System.nanoTime();
        collection.contains((int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testRemove(Collection<Integer> collection) {
        long startTime = System.nanoTime();
        collection.remove((int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testClear(Collection<Integer> collection) {
        long startTime = System.nanoTime();
        collection.clear();
        long endTime = System.nanoTime();
        return endTime - startTime;
    }
```

```java
    // Define the methods for maps.
    public static long testPut(Map<Integer, Integer> map) {
        long startTime = System.nanoTime();
        map.put((int) (Math.random() * 100000), (int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testContainsKey(Map<Integer, Integer> map) {
        long startTime = System.nanoTime();
        map.containsKey((int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testRemoveMap(Map<Integer, Integer> map) {
        long startTime = System.nanoTime();
        map.remove((int) (Math.random() * 100000));
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static long testClearMap(Map<Integer, Integer> map) {
        long startTime = System.nanoTime();
        map.clear();
        long endTime = System.nanoTime();
        return endTime - startTime;
    }

    public static void graphPlot(long[][] avgTimes) {

        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter("Graph.txt"));
            for(long[] arr : avgTimes){
                for(long l : arr){
                    writer.write(String.valueOf(l));
                    writer.write("   ");
                }
                writer.write("\n");
            }
            writer.close();

        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

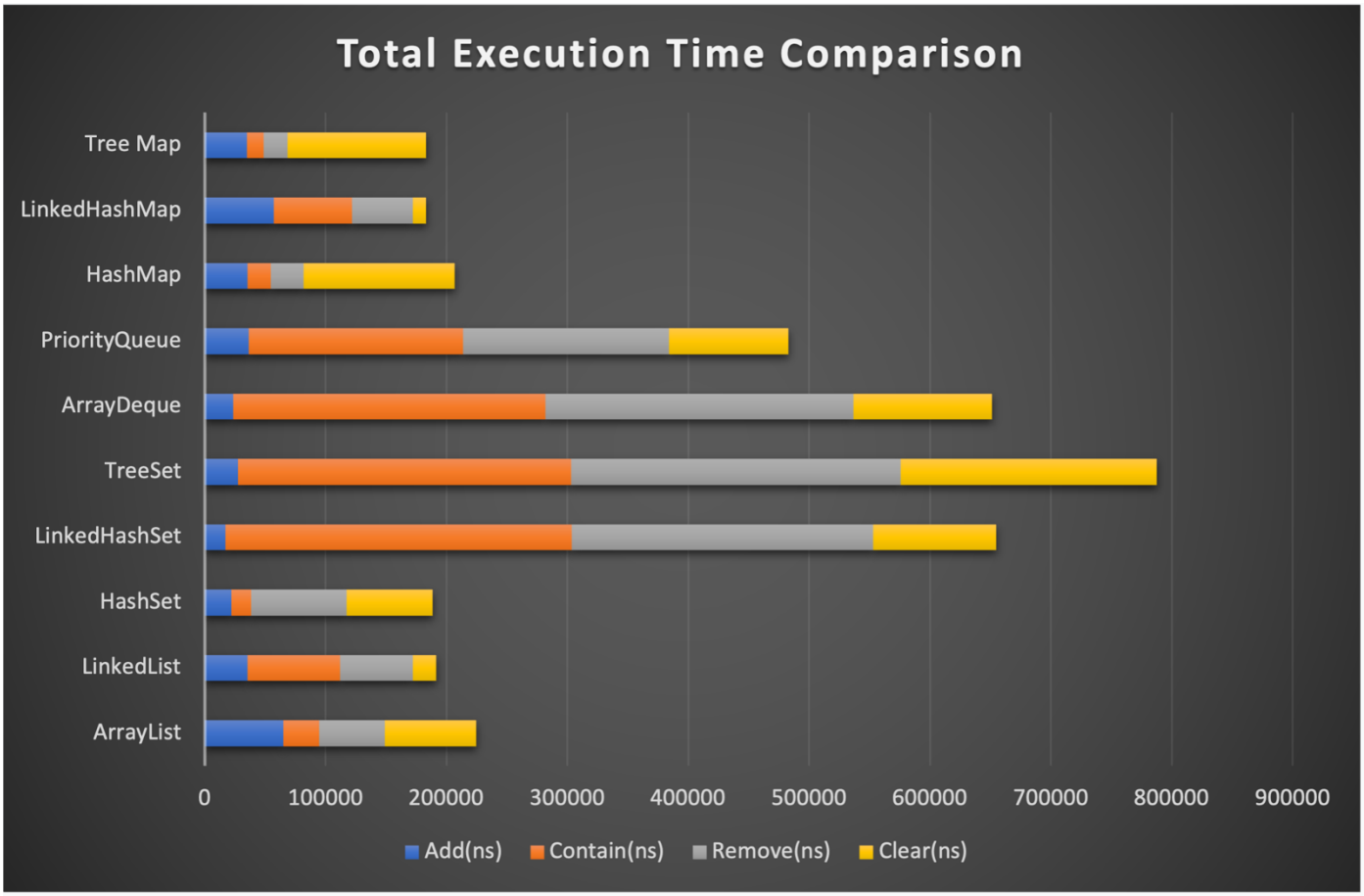
3. Performance Data Table

	Add(ns)	Contain(ns)	Remove(ns)	Clear(ns)
ArrayList	65294	29336	54580	75372
LinkedList	35459	76592	60003	19334
HashSet	22048	16499	79043	70963
LinkedHashSet	17210	286498	249465	101584
TreeSet	27836	275165	272539	211796
ArrayDeque	23875	258250	254626	114417
PriorityQueue	36330	177668	169958	99003
HashMap	35714	19164	27042	124875
LinkedHashMap	57286	64499	50459	10953
Tree Map	35127	13626	19709	114499



Collection Framework — Class Hierarchy


Execution Time Comparison
350000
300000
250000
200000
150000
100000
50000
0
ArrayList
LinkedList
HashSet
LinkedHashSet
TreeSet
ArrayDeque
PriorityQueue
HashMap
LinkedHashMap
Tree Map
Add(ns)
Contain(ns)
Remove(ns)
Clear(ns)


Total Execution Time Comparison
Tree Map
LinkedHashMap
HashMap
PriorityQueue
ArrayDeque
TreeSet
LinkedHashSet
HashSet
LinkedList
ArrayList
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
Add(ns)
Contain(ns)
Remove(ns)
Clear(ns)

4. Discussion

	Add	Contains	Remove	Clear
ArrayList	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(n)	O(n)
HashSet	O(n)	O(1)	O(1)	O(1)
LinkedHashSet	O(1)	O(1)	O(1)	O(1)
TreeSet	O(log(n))	O(log(n))	O(log(n))	O(log(n))
ArrayDeque	O(1)	O(n)	O(n)	O(n)
PriorityQueue	O(log(n))	O(n)	O(n)	O(n)
HashMap	O(1)	O(1)	O(1)	O(1)
LinkedHashMap	O(1)	O(1)	O(1)	O(1)
Tree Map	O(log(n))	O(log(n))	O(log(n))	O(log(n))

4.1. Add

4.1.1. **Syntax:**

Collection.add(Object element)

- The add( ) method in java collections is a fundamental operation for adding elements to various types of collections in java, and it typically returns as boolean value indicating whether the operation was successful.
- Its performance characteristics may vary depending on the specific implementation and the type of collection being used.

4.1.2. **Add method throws 6 exceptions.**

1 NullPointerException : Occurs if you attempt to add a ‘null’ element to a that does not allow ‘null’ values. Ex. HashSet, TreeSet

2 ClassCastException : When you are trying to add an element that is incompatible type to a collection.

3 IllegalArgumentException : Happens when an invalid argument is passed ‘add( )’ method.

4 UnsupportedOperationException : This exception occurs when attempting to add an element to an immutable or read-only collection.

5 ConcurrentModificationException : If a collection is modified while it's being iterated over, typically in multi-threaded scenarios.

6 IllegalStateException : Some collections may throw this exception when attempting to add an element under certain conditions, such as attempting to add an element to a sorted collection like 'TreeSet' when the element violates the ordering constrains.

4.2. Contain

4.2.1. **Syntax:**

Collection.contains(Object element)

- The contains( ) function in Java we use to check whether the element exists or is not in the collection.
- This method returns a boolean value, if the element is present it returns true else it returns false.

4.2.2. Exceptions:

Contains method throws two Exceptions.

1 **) NullPointerException**: The element is null and the collection does not allow null elements.

2)**ClassCastException**: If the type of the specified element prohibits its addition to this collection.

- By considering observations of the performance code, we can say different types of Java collections have different time complexity for the contains() method.

- ArrayList, LinkedList, ArrayDeque, and PriorityQueue require O(n) time. It is depends on the number of items we have in the array. The contains() of

HashSet, LinkedHashSet, HashMap and LinkedHashMap run in O(1) time because determining the object's bucket location requires constant time.

- TreeMap and TreeSet use the divide and conquer approach to check the object value , resulting in a time complexity of O(log n) due to efficient binary search tree structures.

4.3. Remove

4.3.1 **Syntax:**

Collection.remove(Object element)

Or

Collection.remove(Object index)

- The remove() method of Java Collection Interface is used to remove a single instance.
- This method returns a boolean value, if the element is present it returns true else it returns false.
- By considering observations of the performance code, we can say different types of Java collections have different time complexity for the contains() method.

The remove() of HashSet, LinkedHashSet, HashMap and LinkedHashMap run in O(1) time because determining the object's bucket location requires constant time.

ArrayList , LinkedList, ArrayDeque and PriorityQueue has O(n) time complexity

4.4. Clear

4.4.1. **Syntax:**

Collection.clear()

The clear( ) function in Java we use to delete all item in collection. This method returns a boolean value, if the element is present it returns true else it returns false.

4.4.2. **Exceptions:**

Contains method throws two Exceptions.

1 **) Concurrent Modification Exception**: if one thread is iterating over the collection while another thread is trying to clear it. (we can use synchronize key word to handle that exception)

2) **UnsupportedOperationException:** some java collections don't support clear method.

(E.g. -: unmodifiableCollection). We can handle that by unsupported Operation Exception

- By considering observations of the performance code, we can say different types of Java collections have different time complexity for the clear() method.
- TreeSet has an efficient execution time in clear method because of theat classes are implemented as balanced binary search tree.
- Linkedlist has the most execution time because of that need to individually remove each nodes.