PERFORMANCE OF JAVA COLLECTIONS

GROUP NAME : CODE RAPTORS
PROJECT NO : 02
SUBMITTED ON : 17-02-2024

Table of contents

1. Description of Program Design……01

2. Full Java Program Code used for Testing…….02

3. Comparison Table of Performance data……...……13

4. Discussion…….……14

5. Conclusion……18

Description of Program Design

The program is designed to evaluate the performance of different Java Collection implementations across four key methods: add, contain, remove, and clear. The project #2 consists of 3 main processes which are data preparation, testing process, and report creation.

Initially, it generates a dataset comprising 100,000 randomly generated Integer objects, serving as the test elements. Subsequently, the program iterates through each Collection implementation, executing each method multiple times to obtain accurate time measurements. By repeating each operation 100 times and calculating the average time, the program ensures reliable performance evaluations.

Following the execution phase, the program compiles the results into a comprehensive report. This report includes the testing code, a detailed comparison table illustrating the performance data, and brief discussions highlighting the root causes for performance variations such as data structure design, underlying algorithms, and specific use cases where one implementation may outperform others.

These discussions could investigate elements such as the inherent design of data structures, the efficiency of underlying algorithms, and potential optimization strategies.

Full Java Program Code used for Testing

```
import java.util.*;
import java.io.*;

class Project_2{
    // Method to calculate time for operations on Sets
    public static void calculateTimeSets(Set<Integer> set, FileWriter writer) {
        Random rand = new Random();
        // Adding random elements to the set
        for (int i = 0; i < 1000000; i++) {
            int element = rand.nextInt(100000);
            set.add(element);
        }

        // Creating a temporary collection for later use
        Collection<Integer> tempCollection = new ArrayList<Integer>();
        for (Integer num : set) {
            tempCollection.add(num);
        }

        long addTimes = 0;
        long removeTimes = 0;
        long checkTimes = 0;
        long clearTimes = 0;
        int element3 = rand.nextInt(100000);
        // Ensuring element3 is not already in the set
        while (set.contains(element3)) {
            element3 = rand.nextInt(100000);
        }
```

```
// Performing add, contains, and remove operations and measuring time
for (int i = 0; i < 100; i++) {
    int element = rand.nextInt(100000);
    long startTime = System.nanoTime();
    set.add(element);
    long endTime = System.nanoTime();
    addTimes += endTime - startTime;

    int element2 = rand.nextInt(100000);
    startTime = System.nanoTime();
    set.contains(element2);
    endTime = System.nanoTime();
    checkTimes += endTime - startTime;

    startTime = System.nanoTime();
    set.remove(element3);
    endTime = System.nanoTime();
    removeTimes += endTime - startTime;
}

// Clearing the set and adding elements back from the temporary collection
for (int i = 0; i < 100; i++) {
    long startTime = System.nanoTime();
    set.clear();
    long endTime = System.nanoTime();
    clearTimes += endTime - startTime;

    for (Integer number : tempCollection) {
        set.add(number);
    }
}
```

```
    // Writing results to file
    try {
        System.out.println(addTimes / 100 + "\t" + checkTimes / 100 + "\t " + removeTimes / 100 + " \t"
                + clearTimes / 100 + "\n");
        writer.write("\t" + addTimes / 100 + "\t\t" + checkTimes / 100 + "\t\t\t" + removeTimes / 100 + "\t\t"
                + clearTimes / 100 + "\n");
    } catch (IOException e) {
        System.err.println("Error writing to file: " + e.getMessage());
    }
}

// Method to calculate time for operations on Queues
public static void calculateTimeQueue(Queue<Integer> queue, FileWriter writer) {
        Random rand = new Random();
        // Adding random elements to the queue
        for (int i = 0; i < 1000000; i++) {
            int element = rand.nextInt(100000);
            queue.add(element);
        }

        // Creating a temporary collection for later use
        Collection<Integer> tempCollection = new ArrayList<Integer>();
        for (Integer num : queue) {
            tempCollection.add(num);
        }

        long addTimes = 0;
        long removeTimes = 0;
        long checkTimes = 0;
        long clearTimes = 0;
```

```
int element3 = rand.nextInt(100000);
// Ensuring element3 is not already in the queue
while (queue.contains(element3)) {
    element3 = rand.nextInt(100000);
}

// Performing add, contains, and remove operations and measuring time
for (int i = 0; i < 100; i++) {
    int element = rand.nextInt(100000);
    long startTime = System.nanoTime();
    queue.add(element);
    long endTime = System.nanoTime();
    addTimes += endTime - startTime;

    int element2 = rand.nextInt(100000);
    startTime = System.nanoTime();
    queue.contains(element2);
    endTime = System.nanoTime();
    checkTimes += endTime - startTime;

    startTime = System.nanoTime();
    queue.remove(element3);
    endTime = System.nanoTime();
    removeTimes += endTime - startTime;
}

// Clearing the queue and adding elements back from the temporary collection
for (int i = 0; i < 100; i++) {
    long startTime = System.nanoTime();
    queue.clear();
```

```
            long endTime = System.nanoTime();
            clearTimes += endTime - startTime;

            for (Integer number : tempCollection) {
                queue.add(number);
            }
        }

        // Writing results to file
        try {
            System.out.println(addTimes / 100 + "\t" + checkTimes / 100 + "\t " + removeTimes / 100 + " \t"
                    + clearTimes / 100 + "\n");
            writer.write("\t" + addTimes / 100 + "\t\t" + checkTimes / 100 + "\t\t" + removeTimes / 100 + "\t\t"
                    + clearTimes / 100 + "\n");
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    }

// Method to calculate time for operations on Lists
public static void calculateTimeLists(List<Integer> list, FileWriter writer) {
    Random rand = new Random();
    // Adding random elements to the list
    for (int i = 0; i < 1000000; i++) {
        int element = rand.nextInt(100000);
        list.add(element);
    }
    // Creating a temporary list for later use
    List<Integer> tempList = new ArrayList<Integer>();
    for (Integer num : list) {
        tempList.add(num);
```

```
}

long addTimes = 0;
long removeTimes = 0;
long checkTimes = 0;
long clearTimes = 0;

int element3 = rand.nextInt(100000);
// Ensuring element3 is not already in the list
while (list.contains(element3)) {
    element3 = rand.nextInt(100000);
}

// Performing add, contains, and remove operations and measuring time
for (int i = 0; i < 100; i++) {
    int element = rand.nextInt(100000);
    long startTime = System.nanoTime();
    list.add(element);
    long endTime = System.nanoTime();
    addTimes += endTime - startTime;

    int element2 = rand.nextInt(100000);
    startTime = System.nanoTime();
    list.contains(element2);
    endTime = System.nanoTime();
    checkTimes += endTime - startTime;

    startTime = System.nanoTime();
    list.remove(element3);
    endTime = System.nanoTime();
    removeTimes += endTime - startTime;
```

```
    }

    // Clearing the list and adding elements back from the temporary list
    for (int i = 0; i < 100; i++) {
        long startTime = System.nanoTime();
        list.clear();
        long endTime = System.nanoTime();
        clearTimes += endTime - startTime;

        for (Integer number : tempList) {
            list.add(number);
        }
    }

    // Writing results to file
    try {
        System.out.println(addTimes / 100 + "\t" + checkTimes / 100 + "\t " + removeTimes / 100 + " \t"
                + clearTimes / 100 + "\n");
        writer.write("\t" + addTimes / 100 + "\t\t" + checkTimes / 100 + "\t\t" + removeTimes / 100 + "\t\t"
                + clearTimes / 100 + "\n");
    } catch (IOException e) {
        System.err.println("Error writing to file: " + e.getMessage());
    }
}

// Method to calculate time for operations on Maps
public static void calculateTimeMaps(Map<Integer, Integer> map, FileWriter writer) {
    Random rand = new Random();
    for (int i = 0; i < 1000000; i++) {
        int element1 = rand.nextInt(100000);
```

```
    int element2 = rand.nextInt(100000);
    map.put(element1, element2);
}

Map<Integer, Integer> tempMap = new HashMap<Integer, Integer>();
for (Integer key : map.keySet()) {
    int value = map.get(key);
    map.put(key, value);
}

long addTimes = 0;
long removeTimes = 0;
long checkTimes = 0;
long clearTimes = 0;

for (int i = 0; i < 100; i++) {
    int element1 = rand.nextInt(100000);
    int element2 = rand.nextInt(100000);
    long startTime = System.nanoTime();
    map.put(element1, element2);
    long endTime = System.nanoTime();
    addTimes += endTime - startTime;

    int element3 = rand.nextInt(100000);
    startTime = System.nanoTime();
    map.containsValue(element3);
    endTime = System.nanoTime();
    checkTimes += endTime - startTime;

    int element4 = rand.nextInt(100000);
    while (map.containsKey(element4)) {
```

```
            element4 = rand.nextInt(100000);
        }
        startTime = System.nanoTime();
        map.remove(element4);
        endTime = System.nanoTime();
        removeTimes += endTime - startTime;
    }

    for (int i = 0; i < 100; i++) {
        long startTime = System.nanoTime();
        map.clear();
        long endTime = System.nanoTime();
        clearTimes += endTime - startTime;

        for (Integer key : tempMap.keySet()) {
            int value = tempMap.get(key);
            map.put(key, value);
        }
    }

    // Writing results to file
    try {
        System.out.println(addTimes / 100 + "\t" + checkTimes / 100 + "\t " + removeTimes / 100 + " \t"
                + clearTimes / 100 + "\n");
        writer.write(addTimes / 100 + "\t\t" + checkTimes / 100 + "\t\t" + removeTimes / 100 + "\t\t"
                + clearTimes / 100 + "\n");
    } catch (IOException e) {
        System.err.println("Error writing to file: " + e.getMessage());
    }
}
```

```
public static void main(String[] args) {

    String filename = "output.txt"; // Specify the filename
    try {
        FileWriter writer = new FileWriter(filename);
        writer.write("            addTimes   checkTimes   removeTimes  clearTimes\n");
        // HashSet
        Set<Integer> hashSet = new HashSet<>();
        writer.write("HashSet\t\t");
        calculateTimeSets(hashSet, writer);

        // TreeSet
        Set<Integer> treeSet = new TreeSet<>();
        writer.write("TreeSet\t\t");
        calculateTimeSets(treeSet, writer);

        // LinkedHashSet
        Set<Integer> linkedHashSet = new LinkedHashSet<>();
        writer.write("LinkedHashSet");
        calculateTimeSets(linkedHashSet, writer);

        // ArrayList
        List<Integer> arrayList = new ArrayList<>();
        writer.write("ArrayList\t");
        calculateTimeLists(arrayList, writer);

        // LinkedList
        List<Integer> linkedList = new LinkedList<>();
        writer.write("LinkedList\t");
        calculateTimeLists(linkedList, writer);
        // ArrayDeque
```

```
        Deque<Integer> arrayDeque = new ArrayDeque<>();
        writer.write("ArrayDeque\t");
        calculateTimeQueue(arrayDeque, writer);

        // PriorityQueue
        Queue<Integer> prioritryQueue = new PriorityQueue<>();
        writer.write("PriorityQueue");
        calculateTimeQueue(prioritryQueue, writer);

        // HashMap
        Map<Integer, Integer> hashMap = new HashMap<>();
        writer.write("HashMap\t\t\t");
        calculateTimeMaps(hashMap, writer);

        // TreeMap
        Map<Integer, Integer> treeMap = new TreeMap<>();
        writer.write("TreeMap\t\t\t");
        calculateTimeMaps(treeMap, writer);

        // LinkedHashMap
        Map<Integer, Integer> linkedHashMap = new LinkedHashMap<>();
        writer.write("LinkedHashMap\t");
        calculateTimeMaps(linkedHashMap, writer);
        writer.close(); // Close the writer when done
      } catch (Exception e) {
        System.err.println("Error writing to file: " + e.getMessage());
      }

    }
}
```

Comparison Table

Collection	Time Taken for each Method (nanoseconds)			
	add	contains	remove	clear
HashSet	269	343	513	73400
TreeSet	828	777	829	2023
LinkedHashSet	184	385	146	59488
ArrayList	179	107239	337408	235387
LinkedList	315	328727	158602	3262562
ArrayDequeue	519	96929	1278293	461298
PriorityQueue	668	1114904	2454754	316622
HashMap	319	225220	1867	849
TreeMap	771	2558703	3340	356
LinkedHashMap	261	745820	320	1072

Discussion

Various factors can impact the performance variations observed among different Java Collection implementations.

01. Design of Data Structures:

The design of the underlying data structure significantly impacts the performance of Collection operations as it affects the time complexity of different list methods.

02. Algorithms Employed:

The algorithms employed for implementing Collection operations differ across various implementations. This difference may cause performance variations among list methods.

03. Memory Overhead:

Some Collection implementations may have higher memory overhead than others, impacting performance, especially for large datasets. This factor may contribute to performance disparities among list methods

04. Concurrency Requirements:

If the application requires concurrent access to Collection objects from multiple threads, the choice of concurrent Collection implementations such as ConcurrentHashMap or ConcurrentSkipListMap can significantly impact performance due to their thread-safe designs.

Now let's examine the factors influencing performance variations for each Collection separately.

01. HashSet

HashSet provides efficient performance for adding, removing, and checking for elements without duplicates by employing hashing techniques. Its effectiveness depends on the quality of the hash function and the load factor threshold, which may impact performance variations. Additionally, the distribution of elements among hash buckets influences the time complexity of operations such as adding, removing, checking, and clearing of elements.

02. TreeSet

TreeSet is characterized by its balanced binary search tree structure, which ensures elements are kept in a sorted order, providing logarithmic time complexity for operations. It is particularly useful for scenarios requiring orderly traversal and efficient element retrieval.

Performance differences can arise due to the balance of the binary search tree and the effectiveness of tree balancing operations. The size and structure of the tree also play a role in determining the time complexity of operations such as adding, removing, checking and clearing of elements.

03. LinkedHashSet

LinkedHashSet combines the features of HashSet and LinkedList by preserving the order of element insertion while maintaining constant-time performance for basic operations. This makes it suitable for applications requiring both uniqueness and sequential access.

Performance variations may arise due to factors similar to HashSet, with the added consideration of preserving insertion order. Additionally, the overhead of managing the linked list for insertion order can affect the performance of add and remove operations.

04. ArrayList

ArrayList serves as a dynamic array, proving particularly effective in situations requiring frequent random access to elements, although it may experience slower removal time due to array resizing.

Performance disparities may arise due to the chosen strategy for resizing and the cost associated with array copying during resizing operations. The time complexity of add and remove operations is contingent upon the specific position within the array where elements are inserted or deleted.

05. LinkedList

LinkedList operates with a doubly linked list structure, enabling efficient insertion and removal at both ends, although accessing elements by index may be slower compared to ArrayList. It is particularly beneficial for applications focusing on frequent modifications at the extremities of the list.

Performance differences arise from the overhead of managing node references and the traversal cost of the list. The time complexity of add and remove operations varies depending on the position of insertion or deletion within the list.

06. ArrayDeque

ArrayDeque is designed as a resizable-array deque, ensuring constant-time performance for adding and removing elements at both ends, making it suitable for scenarios demanding efficient stack and queue operations.

Performance variations are affected by the circular array structure and the expense of resizing the array. The time complexity of add and remove operations hinges on whether elements are added or removed from the front or back of the deque.

07. PriorityQueue

PriorityQueue organizes elements based on their priority, allowing rapid access to the highest-priority element. It is particularly useful in applications like task scheduling or event-driven systems where prioritized ordering is crucial.

Performance disparities can stem from the effectiveness of the underlying heap structure and the expense of heapifying operations. The time complexity of add and remove operations is contingent on the position of insertion or removal relative to the structure of the heap.

08. HashMap

HashMap utilizes hashing mechanisms for key-value storage, offering consistent performance for fundamental operations such as add, get, remove, and clear. It is commonly used in situations demanding efficient key-based data retrieval and storage.

Performance fluctuations are influenced by the effectiveness of the hash function and the load factor threshold. The arrangement of key-value pairs among hash buckets can affect the time complexity of operations like adding, removing, clearing and checking for elements.

09. TreeMap

TreeMap, like TreeSet, utilizes a red-black tree to maintain elements in sorted order, guaranteeing logarithmic time complexity for operations. It is beneficial for applications requiring organized traversal and retrieval based on keys.

Performance variations are influenced by the balance of the binary search tree and the effectiveness of tree balancing operations. The size and structure of the tree impact the time complexity of operations such as adding, removing, clearing elements and checking for elements.

10. LinkedHashMap

LinkedHashMap combines the features of hashing and linked list, preserving insertion order while providing constant-time performance for essential operations. It is suitable for scenarios demanding predictable iteration order and efficient key-value access.

Performance variations may arise from factors similar to HashMap, with the additional consideration of maintaining insertion order. The overhead of managing the linked list for insertion order can affect the performance of contain operation.

Conclusion

Based on the provided performance data, the conclusions regarding the best and worst collections for each operation are as follows:

In terms of **adding elements** Array List stands out as the top performer, closely trailed by Linked Hash Set and Linked Hash Map. For **checking elements**, HashSet Set excels in efficiency, with LinkedHashSet also providing noteworthy. When it comes to **removing elements**, LinkedHashSet showcases the most efficient operations, while LinkedHashMap and HashSet also exhibit favorable performance. For **clearing elements**, TreeMap emerges as the most efficient option.

TreeSet demonstrates the longest times for adding elements, suggesting its lack of efficiency in this operation. TreeMap and PriorityQueue exhibit the poorest performance in terms of checking for element existence, while PriorityQueue also demonstrates inefficient performance in removing elements. LinkedList displays the most inefficient performance in clearing elements.

These findings emphasize the significance of selecting the appropriate data structure based on the specific requirements and usage patterns of the application to achieve optimal performance.

Project Members:

1. Amarasinghe A.A.D.H.S. – 220023U
2. Ranasinghe P.I. – 220503R
3. Rathnayaka R.L.M.P. – 220520P
4. Samarakoon E.S.P.A. – 220548H