Performance of several Java Collection implementations

By CodeCraft

Group Members

- Wijesooriya J.M.I – 220724U
- Gallage A.H. – 220172A
- Wijeratne E.N.K – 220717C
- Weerasiri M.K.S.L – 220689N

Table of Contents

1.Program Description

The aim of our program is to implement several Java Collection data structures, and test their performance for a few methods. It consists of implementations of 10 different data structures: HashSet, TreeSet, LinkedHashSet, ArrayList, LinkedList, ArrayDeque, PriorityQueue, HashMap, TreeMap, and LinkedHashMap.

The program measures the time taken by each data structure to perform the following operations: adding elements, checking for containment, removing elements, and clearing the entire structure. For each data structure, it performs the four operations 100 times and records the average time taken for each operation.

Considering the implementation of the testing of the methods, the add method adds 100,000 random integers to the data structure. The contains method checks whether each data structure contains -1, which it doesn't, so this is the worst case scenario for contains. The remove method removes an element that was added halfway through the add method, and the clear method clears the data structure completely.

The results are then stored and displayed on the command line, showing the average time taken for each operation for each data structure.

In summary, our program provides a comparative analysis of the performance of different Java data structures under various operations, aiding in understanding their efficiency and suitability for specific use cases.

2.Java program code used for testing

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to
change this license
 */

package com.mycompany.project_02;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

public class Project_02 {
        private static MyDataCollections mdc;
    /*
    HashSet
    TreeSet
    LinkedHashSet
    ArrayList
    LinkedList
    ArrayDeque
    PriorityQueue
    HashMap
    TreeMap
```

```
    LinkdHashMap


    add
    contains
    remove
    clear
    data -> 0-100 000
    time -> ns
    average of  100
    */
    // 2D array to store time measurements
    private static long[][] time=new long[10][4];
    private static String[]
names={"HashSet","TreeSet","LinkedHashSet","ArrayList","LinkedList","ArrayDeque",
"PriorityQueue","HashMap","TreeMap","LinkdHashMap"};

    public static void main(String[] args) {
        long
start,end,duration_add=0,duration_contain=0,duration_remove=0,duration_clear=0;
         // Array of MyDataCollections instances
        MyDataCollections[] collection={new MyHashSet(),new MyTreeSet(),new
MyLinkedHashSet(),new MyArrayList(),new MyLinkedList(),new MyArrayDeque(),new
MyPriorityQueue(),new MyHashMap(),new MyTreeMap(),new MyLinkedHashMap()};
        // Iterating through each collection type
        for(int i=0;i<10;i++){
            for(int j=0;j<100;j++){
            // Timing the 'add' operation
            start=System.nanoTime();
            collection[i].add();
            end=System.nanoTime();
            duration_add+=end-start;
             // Timing the 'contains' operation
            start=System.nanoTime();
            collection[i].contains();
            end=System.nanoTime();
            duration_contain+=end-start;
            // Timing the 'remove' operation
            start=System.nanoTime();
            collection[i].remove();
            end=System.nanoTime();
            duration_remove+=end-start;
            // Timing the 'clear' operation
            start=System.nanoTime();
            collection[i].clear();
```

```
            end=System.nanoTime();
            duration_clear+=end-start;
            }
            // Calculating average time for each operation
            time[i][0]=duration_add/100;
            time[i][1]=duration_contain/100;
            time[i][2]=duration_remove/100;
            time[i][3]=duration_clear/100;

        }
        // Printing the time measurements
        printTime();
    }
    // Method to print time measurements
    public static void printTime(){
        System.out.print("Type\t\tAdd\t\t\tContains\tRemove\t\tClear\t\t\n");
        for(int i=0;i<10;i++){
            if(i==0 || i==1 ||i==7||i==8){
                System.out.print(names[i]+"\t\t");
            }else{
                System.out.print(names[i]+"\t");
            }
            for(int j=0;j<4;j++){
                System.out.print(time[i][j]+"\t\t");
            }
            System.out.println();
        }
    }
    // Abstract class representing data collections
    public static abstract class MyDataCollections{
        public abstract void add();
        public abstract void contains();
        public abstract void remove();
        public abstract void clear();
    }
    // Class representing HashSet
    public static class MyHashSet extends MyDataCollections{
        private Set<Integer> my_hash_set;
        private int removing_ele;
        public MyHashSet(){
            my_hash_set=new HashSet();
        }
        // Adding elements to HashSet
        public void add(){
            for(int i=0;i<100000;i++){
```

```
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_hash_set.add(value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        // Checking if an element exists in HashSet
        public void contains(){
            //check for worst case
            my_hash_set.contains(-1);
        }
        // Removing an element from HashSet
        public void remove(){
            my_hash_set.remove(removing_ele);
        }
        // Clearing HashSet
        public void clear(){
            my_hash_set.clear();
        }
    }
    // Similarly implement classes for other data structures...
    public static class MyTreeSet extends MyDataCollections{
        private Set<Integer> my_tree_set;
        private int removing_ele;
        public MyTreeSet(){
            my_tree_set=new TreeSet();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_tree_set.add(value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_tree_set.contains(-1);
        }
        public void remove(){
            my_tree_set.remove(removing_ele);
```

```
        }
        public void clear(){
            my_tree_set.clear();
        }
    }

    public static class MyLinkedHashSet extends MyDataCollections{
        private Set<Integer> my_linked_hash_set;
        private int removing_ele;
        public MyLinkedHashSet(){
            my_linked_hash_set=new LinkedHashSet();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_linked_hash_set.add(value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_linked_hash_set.contains(-1);
        }
        public void remove(){
            my_linked_hash_set.remove(removing_ele);
        }
        public void clear(){
            my_linked_hash_set.clear();
        }
    }

    public static class MyArrayList extends MyDataCollections{
        private List<Integer> my_array_list;
        private int removing_ele;
        public MyArrayList(){
            my_array_list=new ArrayList();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_array_list.add(value);
```

```
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_array_list.contains(-1);
        }
        public void remove(){
            my_array_list.remove(50000);
        }
        public void clear(){
            my_array_list.clear();
        }
    }

    public static class MyLinkedList extends MyDataCollections{
        private List<Integer> my_linked_list;
        private int removing_ele;
        public MyLinkedList(){
            my_linked_list=new LinkedList();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_linked_list.add(value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_linked_list.contains(-1);
        }
        public void remove(){
            my_linked_list.remove(50000);
        }
        public void clear(){
            my_linked_list.clear();
        }
    }
```

```
    public static class MyArrayDeque extends MyDataCollections{
        private Queue<Integer> my_array_deque;
        private int removing_ele;
        public MyArrayDeque(){
            my_array_deque=new ArrayDeque();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_array_deque.add(value);
                if(i==500){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_array_deque.contains(-1);
        }
        public void remove(){
            my_array_deque.remove(removing_ele);
        }
        public void clear(){
            my_array_deque.clear();
        }
    }

    public static class MyPriorityQueue extends MyDataCollections{
        private Queue<Integer> my_priority_queue;
        private int removing_ele;
        public MyPriorityQueue(){
            my_priority_queue=new PriorityQueue();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_priority_queue.add(value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
```

```
            //check for worst case
            my_priority_queue.contains(-1);
        }
        public void remove(){
            my_priority_queue.remove(removing_ele);
        }
        public void clear(){
            my_priority_queue.clear();
        }
    }

    public static class MyHashMap extends MyDataCollections{
        private Map<Integer,Integer> my_hash_map;
        private int removing_ele;
        public MyHashMap(){
            my_hash_map=new HashMap();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_hash_map.put(i,value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_hash_map.containsValue(-1);
        }
        public void remove(){
            my_hash_map.remove(50000);
        }
        public void clear(){
            my_hash_map.clear();
        }
    }

    public static class MyTreeMap extends MyDataCollections{
        private Map<Integer,Integer> my_tree_map;
        private int removing_ele;
        public MyTreeMap(){
            my_tree_map=new TreeMap();
        }
```

```
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_tree_map.put(i,value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_tree_map.containsValue(-1);
        }
        public void remove(){
            my_tree_map.remove(50000);
        }
        public void clear(){
            my_tree_map.clear();
        }
    }

    public static class MyLinkedHashMap extends MyDataCollections{
        private Map<Integer,Integer> my_linked_hash_map;
        private int removing_ele;
        public MyLinkedHashMap(){
            my_linked_hash_map=new LinkedHashMap();
        }
        public void add(){
            for(int i=0;i<100000;i++){
                Random ran=new Random();
                int value=Math.abs(ran.nextInt()%100000);
                my_linked_hash_map.put(i,value);
                if(i==50000){
                    removing_ele=value;
                }
            }
        }
        public void contains(){
            //check for worst case
            my_linked_hash_map.containsValue(-1);
        }
        public void remove(){
            my_linked_hash_map.remove(50000);
        }
```

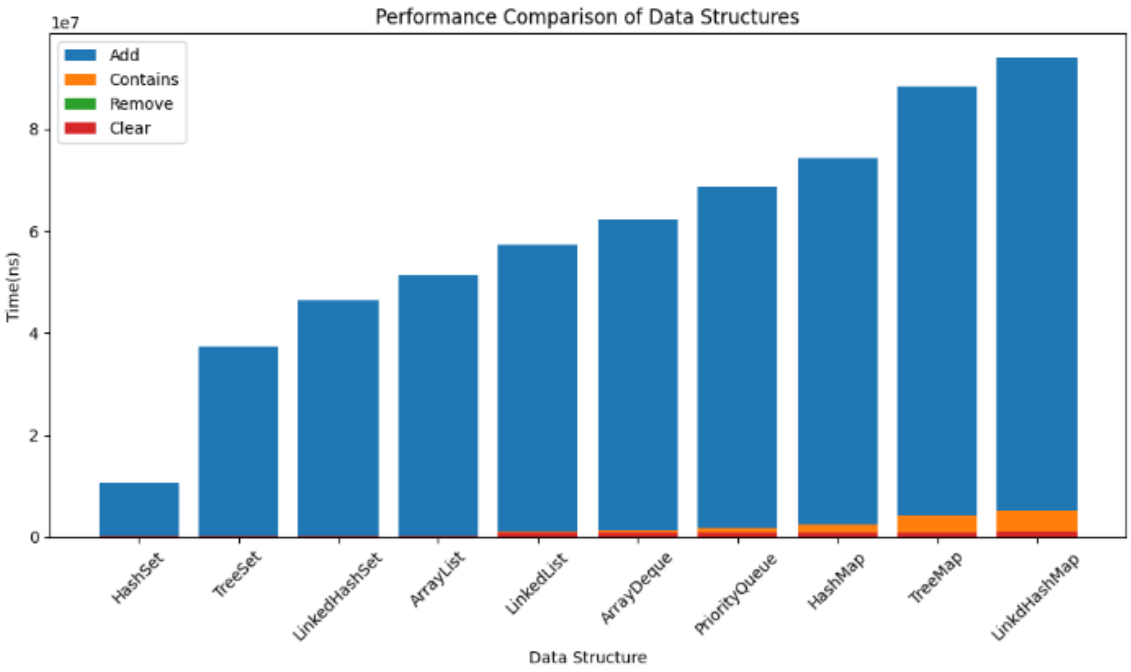
```
        public void clear(){
            my_linked_hash_map.clear();
        }
    }


}
```

3. Comparison table of performance data

The below table depicts the results obtained from the performance of several Java Collection implementations, focusing on methods such as, Add, Contains, Remove and Clear. These evaluations were carried out under standard conditions, utilizing default initialCapacity and loadFactor settings. Each collection was tested using 100,000 randomly generated Integers. The time is measured in nanoseconds. To ensure accuracy and reliability, each test was done 100 times, and reported figures represent the calculated averages. This was performed using NetBeans.

Type	Add(ns)	Contains(ns)	Remove(ns)	Clear(ns)
HashSet	10705596	11310	5421	114620
TreeSet	37450713	22380	12217	115415
LinkedHashSet	46353136	31347	19616	156246
ArrayList	51521044	317275	42231	189752
LinkedList	57217799	1056588	183945	810941
ArrayDeque	62255536	1355226	209838	886665
PriorityQueue	68758599	1875989	311985	925054
HashMap	74220820	2566699	316189	969390
TreeMap	88311044	4387491	329346	969805
LinkdHashMap	94010917	5318987	332386	1040577



4. Brief descriptive discussions on the reasons for performance variations of above results.

The performance variations in the above data can vary because of several factors, including the data structure's inbuilt design, the characteristics of the each operation being performed, and the nature of the data .

4.1. Data Structure Design:

Different data structures have different designs and implementations, which affect their performance. For example, HashSet uses a hash table, while Tree Set uses a balanced binary search tree. Because of this fundamental difference we can see variations in performance for operations such as add, remove, and contains, clear.

4.2. Operation Complexity:

Each operation (add, remove, contains, clear) has its own time complexity, and this complexity can vary significantly between data structures. For instance, adding an element to an ArrayList is O(1) on average, but if the ArrayList needs to resize, it is O(n)(n is size of the array).Because if ArrayList has to resize then program has to build a whole new object. Similarly, adding an element to a TreeSet or TreeMap more complex compared to HashSet or HashMap.

4.3. Nature of Data:

The performance of above operations can be different to one another by the nature and distribution of the data which we use. For example, if the data being added to a HashSet or HashMap causes frequent hash collisions, it can degrade performance due to increased chaining or tree height in the data structure. Operations on sorted data may perform differently compared to unsorted data, affecting structures like TreeSet and TreeMap.

4.4. Internal Implementations:

The specific optimizations and algorithms used in the implementation of each data structure can also affect on performance. Such as, different ways for handling collisions in hash tables can lead to variations in performance for HashSet and HashMap.

4.5. Size of the Data Structure:

The size of the data structure can also affect performance, especially for operations like add and remove. Some data structures may require resizing or rebalancing as they grow, leading to increased time complexity for these operations as the size of the structure increases.

Program Code - https://drive.google.com/file/d/1DFRkeHAFeXffiR9MPvx5sYPo5Tcpe0-Y/view?usp=sharing