**Sorted Maps and Subsequent Testing: A Full Report**

By: Yifan Xu (E Block) and Bhavik Nagda (C Block)

**Table Of Contents**

|  |  |
| --- | --- |
| **Topics** | **Page Numbers** |
| Introduction | 2 |
| Expected Big-O Values | 2 |
| Our Tests | 3 |
| Our Benchmarks | 4 |
| Adjusting the Big-O Values for our Performance Data | 4 |
| Our Benchmark Results | 5 |
| Expected vs. Actual Performance | 7 |
| Results Analysis | 9 |
| Evaluation of Benchmark | 9 |
| Appendix A: Test and Benchmark Code | 10 |
| Appendix B: SortedMap Code | 12 |
| Appendix C: Member Contributions | 14 |
|  |  |

**Introduction**

The assigned project involves the creation and subsequent testing of the SortedMap class in the Java programming language. Java’s maps are data structures that allow us to assign a piece of data as a key to another piece of data. They are particularly useful in associating two types of data, and matching two different ideas one to one. To improve our understanding of maps, we were previously required to code an UnsortedMap class, one in which the keys and the values were unsorted. The UnsortedMap class is implemented using two arrays; the index for the key array (storing the key objects) pointed to data in the value array (which stored value objects). Essentially, the arrays were ‘parallel.’ This previous project also involved a cMap interface that was implemented by UnsortedMap.

The project covered in this report not only requires us to create a SortedMap class, but it also requires us to test for the efficiency of this map implementation. A sorted map involves a sorted keys array, and we can thus use binary searches on this keys array to get a value. The first half of the following report will cover the efficiency of different implementations (using Big-O notation) and testing, while the second half will cover benchmarking (the actual measurement of the efficiency of the maps using timers and large data). The specific code is presented in the appendix. Enjoy!

**Expected Big-O Values**

**Unsorted Map**

**Put: n**

**Explanation:** Putting an object into this unsorted array will first require a for loop cycling through the array to check if the object already exists. This first loop makes O(n) = n for this operation. If the object already exists, then the value in the value array will be changed, which takes constant time. If the object doesn’t already exist, then the key and value are appended to the end of the keys and values arrays, which also takes constant time.

**Get: n**

**Explanation:** Getting a value from a given key will require a for loop cycling through the keys array to find the index of the key itself. Then this index will be used to trace the value in the value array. Since we are looping through the array, this operation will have a O(n) of n.

**Sorted Map**

**Put: n**

**Explanation:** Putting an object into this sorted array will first require a for loop cycling through this sorted array to find the proper location for the comparable object. Then, after this location is found, every object below the location in the array must be moved down. So we first work with every element of the upper part above the location (in a for loop), then we work with every element of the lower part below the location (to shift the data downward). Thus, since we work with nearly every element of the array, O(n) = n.

**Get: log(n)**

**Explanation:** Note that the keys array is already sorted by object. Thus, to find the index within the keys array, only a binary search is required. Now, a binary search by definition ‘cuts’ the array in half (it checks if the value is greater or less than the middle value in the array, and chooses the upper or lower half of the array accordingly) every single operation. It thus takes log2n, or log(n) time to find the key index, and then this index is used to find the corresponding value, requiring constant time. Thus, in total, O(n) = log(n).

**Our Tests**

We used the test that Mr. Stulin gave us for the UnsortedMap project to test the SortedMap class and to verify the proper working of the SortedMap. Here’s how this test works:

1. An String array called *stringList* of size 10,000 is created, and a random String sequence with 10 characters is added in every spot of the array (10,000 random strings, each made of 10 characters, are added to a String array, with one string in every spot).
2. The SortedMap is created also of size 10,000, and each element of *stringList* becomes a key in this SortedMap using the put operation, with the respective value as the uppercase representation of the key.
3. A for- loop runs through the entire SortedMap (using the indices), and checks to make sure that the value for each respective key is the uppercase representation of the key (this uses the get operation).
4. If any problem occurs with any of these steps, either an error is thrown or the program reads “Map Bad.” If no problems occur and everything checks out, then the system reads “Map OK”

We thus ran this tester method using our SortedMap (instead of the UnsortedMap), and the system read “Map OK,” indicating that our implementation of the Sorted Map works. We believe that this test is appropriate because it efficiently and realistically checks to ensure that both put and get work in an integrated process. If either don’t work, then multiple flags are thrown, so we know that our Maps actually work.

**Our Benchmark**

A benchmark tests the efficiency of the program starting and stopping specific timers when necessary. We used basically the exact same benchmarking code that Mr. Stulin gave us. The timer runs from the beginning of the put operations to the end, and prints out the time required to complete the put operations. The timer also runs from the beginning of the get operations to the end, and prints out the time required to complete the get operations (by subtracting the start time from the end time).

Our experimental design lends validity and integrity to our results. For precision, we measured our time in milliseconds, and took averages when computing data points to ensure accuracy. We used another class to print out multiple tests that gave us multiple data points instead of just one at a time. By analyzing the benchmark times, we can evaluate whether or not the Big-O theory matches reality.

**Adjusting the Big-O Values for our Performance Data**

Our test method and benchmarking, as explained previously shows the time needed to perform a large number of put or get operations together. In other words, it doesn’t show the time needed to perform one single put or get operation. We chose not to measure the time it takes to do one extra operation because the time we get may fluctuate a lot and we won’t be able to determine how close our data is to the expected Big O.

Instead, we measured the time it took to measure n put operations together (in our benchmarking), but each of these individual put operations took different times. This is because an earlier put operation may require much less time than a later put operation. Thus, we technically added all of the individual times up to get our measurements, lending validity and most importantly accuracy to our investigation. By adding them up, the individual fluctuations will not affect our data so much since the negative fluctuations cancel out the positive ones.

Now to calculate the relationship between our results and the Big-O, we need to look at some higher level mathematics- Calculus. Since we measured the sum of these individual operations in our benchmarking, we effectually integrated over the individual operation times:

Look familiar? Indeed, since we are measuring the time for n operations added together, the Big-O for our performance data shall be the integral of the Big-O for one operation. Specifically, n becomes n2/2, which is equivalent to n2 when we are talking about Big-O . We thus performed quadratic regressions for the Unsorted Map put and Sorted Map put operations, since the O(n) for that data was n2.

(This new paragraph in red is what we changed from the original)

The get operations are different. Since get operations don’t change the size of the array, each get operation should theoretically take the same amount of time. To find the Big-O for n operations added together, we simply multiply the original Big-O by n. So for Unsorted map get, the new Big-O is n2 , so we performed a quadratic regression for it. For the Sorted Map get operation, the expected Big-O would b n\*log(n), so we performed a transformation on the data:

.

Thus we see that by dividing our performance data by the log of the array size, we are given a linear relationship. Our graph for the Sorted Map get operation was thus adjusted (we divided the data by log of array size) and we looked for a linear regression.

**Our Benchmark Results**

**Figure 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array Size** | **Unsorted Map Times (milliseconds)** | | | | | |
| **The Put Operation** | | | | | |
| **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Average** |
| **10000** | 520 | 417 | 549 | 416 | 407 | 462 |
| **15000** | 1351 | 1294 | 1455 | 1363 | 980 | 1289 |
| **20000** | 2706 | 3361 | 2697 | 1900 | 3609 | 2855 |
| **25000** | 5587 | 2637 | 5002 | 5765 | 2666 | 4331 |
| **30000** | 9052 | 8598 | 3730 | 9096 | 8692 | 7828 |

**Figure 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array Size** | **Unsorted Map Times (milliseconds)** | | | | | |
| **The Get Operation** | | | | | |
| **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Average** |
| **10000** | 109 | 92 | 97 | 101 | 94 | 99 |
| **15000** | 178 | 171 | 240 | 189 | 178 | 191 |
| **20000** | 320 | 352 | 342 | 334 | 328 | 335 |
| **25000** | 504 | 495 | 523 | 457 | 505 | 497 |
| **30000** | 1059 | 705 | 712 | 679 | 758 | 783 |

**Figure 3**

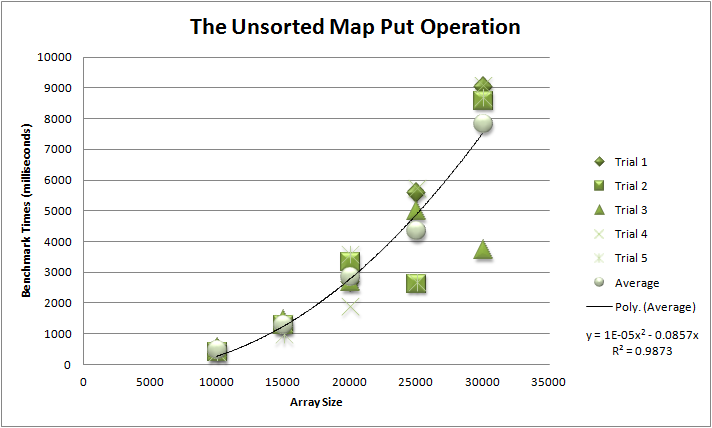
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array Size** | **Sorted Map Times (milliseconds)** | | | | | |
| **The Put Operation** | | | | | |
| **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Average** |
| **10000** | 671 | 265 | 235 | 216 | 271 | 331 |
| **15000** | 1126 | 527 | 864 | 512 | 1017 | 809 |
| **20000** | 1626 | 1093 | 1670 | 1707 | 916 | 1402 |
| **25000** | 2757 | 1638 | 3216 | 2863 | 1738 | 2442 |
| **30000** | 4484 | 3234 | 4721 | 4456 | 2557 | 3890 |

**Figure 4**

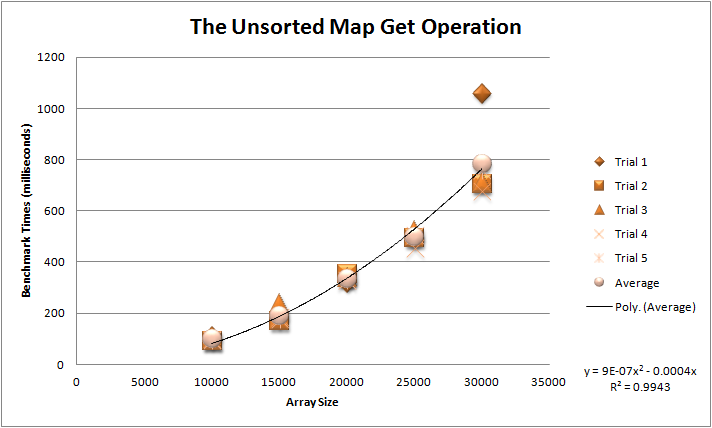
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array Size** | **Sorted Map Times (milliseconds)** | | | | | |
| **The Get Operation** | | | | | |
| **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Average** |
| **10000** | 21 | 7 | 10 | 6 | 6 | 10 |
| **15000** | 13 | 14 | 13 | 18 | 15 | 15 |
| **20000** | 21 | 19 | 25 | 24 | 29 | 24 |
| **25000** | 36 | 20 | 26 | 29 | 28 | 28 |
| **30000** | 35 | 38 | 24 | 52 | 52 | 40 |

**Expected vs. Actual Performance**

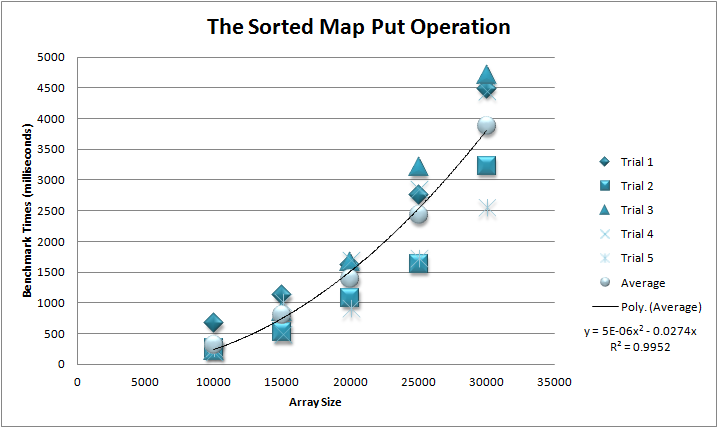
**Figure 5.** A graph of our Unsorted Map Put Operation, and a quadratic regression on the average of the trials (Explanation for the quadratic regression is in the section Adjusting the Big-O Values for our Performance Data).

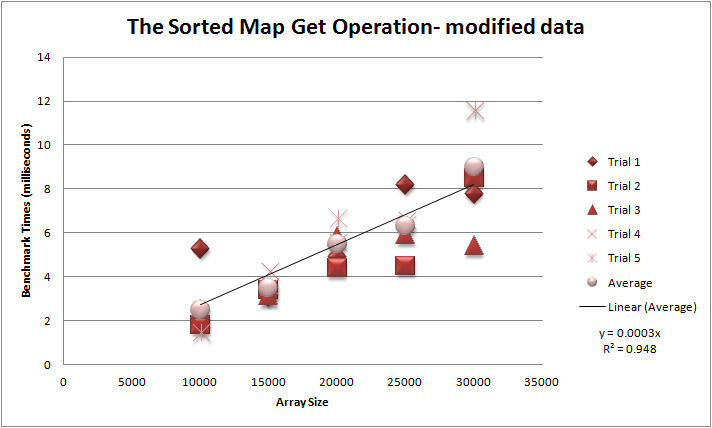


**Figure 6.** A graph of our Unsorted Map Get Operation, and a quadratic regression on the average of the trials (Explanation for this regression is in the section Adjusting the Big-O Values for our Performance Data).



**Figure 7.** A graph of our Sorted Map Put Operation, and a quadratic regression on the average of the trials (Explanation for this regression is in the section Adjusting the Big-O Values for our Performance Data).

**Figure 8.** A graph of our Sorted Map Get Operation, and a linear regression on the average of the trials divided by the log of the array size (Explanation for these adjustments is in the section Adjusting the Big-O Values for our Performance Data).



**Results Analysis**

First, a brief explanation: For each operation on each map we tested with array (filled) size from 10,000 to 30,000 in 5,000 increments, leaving our array size itself at 80,000. We did each of these tests five times in five different trials to ensure the accuracy of our results. The results are shown in figures 1-4. We then plotted the results on a scatter graph as shown in figures 5-8, and calculated the respective regressions necessary that match the Big-O values. The equations for the regressions are shown on each of the graphs themselves. Along with the equations are r2 values, which measure the average deviation from regression. These values will be used in our benchmark evaluation.

One factor contributing to the differences between our expectations and the performance results could have been extra, unnecessary operations that were included as errors in our code. If our code was not as efficient as possible and instead performed unnecessary operations, the inefficiency may have contributed to extra time and greater deviation from the regressions. However, there was one significant factor that could have skewed results that we avoided: the lengthening of an array mid-testing. Indeed, our program includes code that automatically lengthens the array when the array gets full. This requires the creation of a new, larger array and the copying over of all data from the first array to the second. These operations ultimately add extra time that skew results with the benchmarks. We purposely created a large array of size 80,000 when completed our testing to ensure that this would not be a factor contributing to errors in our results.

**Evaluation of Benchmark**

We believe that our Benchmark is effective at collecting the data. The purpose of using a different benchmark is to minimize the influence of fluctuating times. We measured the total time it took for a big number of operations many times and took the average. The effect is significant. As you can see in the graph, although there may be one or two trials that divert a lot from the predicted regression, the average of 5 trials lies really close to the line of best fit. The high R2 value also shows us that our results are very close to the predicted Big-O.

The unsorted map put operation, the unsorted map get operation, and the sorted map put operation were all predicted to have an efficiency of n2 (see Adjusting Big-O values section). This would require a quadratic relationship, and we thus graphed a quadratic regression with a y intercept of zero (since no spots to fill would theoretically take no time). One interesting results was that the data points weren’t all the same; indeed, there were differences in each trial, suggesting that the computer itself had some error/computing time differentials. Furthermore, the larger the data became, the more spread out the benchmark times became. This confirms that the relationship is a polynomial with order of magnitude greater than 1, and the larger the data becomes, the greater the time differentials become as well.

The sorted map get operation was predicted to have an efficiency of n\*log(n), which meant that y/log(n) will give us a linear relationship. And indeed that was what the graph showed us.

The biggest weakness of our benchmark is that when a lot of put operations has the key value that already exist in the map it’s going to affect our integral, because those put operations don’t increase the size of the array, thus should not be a part of our integral. However this is not that big of a problem because we are selecting our data randomly from an extremely giant pool of random elements. There are 2610 different elements to be randomly picked, and the chance that no repeat will happen over the 10000 picks are more than 99.99%. So this is a small problem. Even if repeats does happen, our relation will still be quadratic, it’s just the bounds of the integrals will change a little bit.

**Appendix A: Test and Benchmark Code**

private String stringList[];

private final String chars = "abcdefghijklmnopqrstuvwxyz";

private final Random r = new Random();

private final int MAP\_SIZE = 80000;

private final int MAX\_SIZE = 80000;

private int NUMBER\_GETS = 35000;

private int NUMBER\_PUTS = 35000;

private long startTaskms;

private long endTaskms;

private long a;

private long b;

float firstResult;

float secondResult;

public void startTimer(){

startTaskms = System.currentTimeMillis();

}

public void stopTimer(){

endTaskms = System.currentTimeMillis();

}

public long elapsedTimems(){

return (endTaskms-startTaskms);

}

public long elapsedTimeSeconds(){

return (endTaskms-startTaskms)/1000;

}

public long elapsedTimeTenthsSeconds(){

return (endTaskms-startTaskms)/100;

}

public void benchmarkMessageTenths (String mess){

System.out.println(mess+"took "+elapsedTimems()+" ms");

}

private String makeString (int length){

String s = "";

int numberChars = chars.length();

for (int i = 0;i<length;i++){

s +=chars.charAt(r.nextInt(numberChars));

}

return s;

}

private void makeStringList (int size){

stringList = new String [size];

for (int i = 0;i<size;i++){

stringList[i]=makeString (10);

}

}

public boolean doTest(){

makeStringList (MAX\_SIZE);

String value;

String key;

cMap map = new UnsortedMap (MAX\_SIZE);

startTimer();

for (int i = 0;i<NUMBER\_PUTS;i++){

map.put(stringList[i], stringList[i].toUpperCase());

}

stopTimer();

benchmarkMessageTenths (NUMBER\_PUTS+" put operations ");

a = elapsedTimems();

startTimer();

for (int i = 0;i<NUMBER\_GETS;i++){

key = stringList[i];

value = (String)map.get(stringList[i]);

if (!value.equals(key.toUpperCase())){

return false;

}

}

stopTimer();

b = elapsedTimems();

benchmarkMessageTenths (NUMBER\_GETS+" get operations ");

time timeLa = new time(a,b);

timeList.add(timeLa);

return true;

}

public void timeStats(){

for (time t : timeList){

firstResult+=t.putTime;

secondResult+=t.getTime;

}

firstResult = firstResult/(timeList.size());

secondResult = secondResult/(timeList.size());

System.out.println("for "+timeList.size()+" trials, the put time average is "+firstResult+" ms");

System.out.println("for "+timeList.size()+" trials, the get time average is "+secondResult+" ms");

}

public static void main(String[] args) {//the main method here runs the test

for (int i = 10000;i<35000;i+=5000){

UnsortedMap labRat = new UnsortedMap();

labRat.NUMBER\_GETS=i;

labRat.NUMBER\_PUTS=i;

labRat.doTest();

labRat.doTest();

labRat.doTest();

labRat.doTest();

labRat.doTest();

System.out.println("when number of puts and gets are "+i);

labRat.timeStats();

if (labRat.doTest()){

System.out.println("map OK");

}else{

System.out.println("map BAD");

}

}

}

**Appendix B: SortedMap Code**

Comparable[] keys;

Object[] values;

int counter=0;

static Object[] intermediate;

ArrayList <time> timeList = new ArrayList();

public static boolean contains (Comparable questionable[], Comparable question){

for (int i = 0; i < questionable.length;i++){

if (questionable[i]==question){

return true;

}

}

return false;

}

public void insert (Object array[],Object thing,int a){

for (int i = counter;i>a;i--){

array[i]=array[i-1];

}

array[a]=thing;

}

public SortedMaps (int iniSize){

keys = new Comparable [iniSize];

values = new Object [iniSize];

}

public SortedMaps(){

}

@Override

public void put(Comparable keyValue,Object valueValue){

if (Arrays.binarySearch(keys,0,counter, keyValue)>0){

int a = binarySearch (keys,keyValue);

values[a]=valueValue;

}else{if(counter==keys.length){

keys=Arrays.copyOf(keys, counter + (int) (counter/5));

values = Arrays.copyOf(values,counter + (int) (counter/5));

}

if (counter ==0){

keys[0]=keyValue;

values[0]=valueValue;

counter++;

}

else if (counter==1){

if (keys[0].compareTo(keyValue)>0){

keys[1]=keys[0];

keys[0]=keyValue;

values[1]=values[0];

values[0]=valueValue;

}

if (keys[0].compareTo(keyValue)<0){

keys[1]=keyValue;

values[1]=valueValue;

}

counter++;

}

else {

if (keyValue.compareTo(keys[0])<0){

insert(keys,keyValue,0);

insert(values,valueValue,0);

}

else if (keyValue.compareTo(keys[(counter-1)])>0){

keys[counter]=keyValue;

values[counter]=valueValue;

}else{

int startIndex = 0;

int endIndex = counter-1;

int check = (startIndex+endIndex)/2;

while ((endIndex-startIndex)!=1){

if (keyValue.compareTo(keys[check])>0){

startIndex = check;

check = (startIndex+endIndex)/2;

}

if (keyValue.compareTo(keys[check])<0){

endIndex = check;

check = (startIndex+endIndex)/2;

}

}

insert (keys,keyValue,endIndex);

insert (values,valueValue,endIndex);

}

counter++;

}

}

}

@Override

public Object get(Comparable keyValue){

int a = Arrays.binarySearch(keys,0,counter, keyValue);

if (a>=0){

return values[a];

} else{

return null;

}

}

**Appendix C: Member Contributions**

Ethan wrote half of the code and Bhavik wrote the other half (specifically, Ethan worked on the data collecting class and support methods while Bhavik works on the SortedMap class). As for the report, Ethan wrote the first half up to the benchmarking sections, while Bhavik wrote the latter half. Both Ethan and Bhavik worked on the analysis and regressions together using their mathematical skills and their ingenious ideas. Mr. Stulin also contributed a large part to this report, by providing testing methods and other code and, of course, pressuring students :). (Special thanks for Mr. Stulin for waiting patiently every time when Ethan starts his question with a ‘wait’)