

CSE374

Nicholas Gincley

Program 6

Part 1 Overview

The purpose of this program is to design a standard set class with an underlying data structure to implement its operations efficiently. We are specifically looking to maximize the efficiency of the add, remove and contains methods all of which should perform sub linearly.

For this set program I chose to implement a binary search tree as my underlying data structure. I chose this because it is a fairly simple structure that I am not only very familiar with but it is generally efficient in the way it handles data with an average time of $O(\log(n))$ for most of the basic operations at $O(\log(n))$ in the worst case. Because of this can keep the cost of our key operations like add remove and contains to a minimum and will strive to reach $O(\log(n))$ time with these operations. I did also however implement an ArrayList for certain aspects of my program, specifically the union, difference and intersection methods because it would make those operations much simpler to complete and still run in good time. This was done by, whenever one of those methods was called, a set specific arraylist would be filled with all the values of the tree and then compared to another created arraylist of another set made in the same process to find the intersections, differences etc.

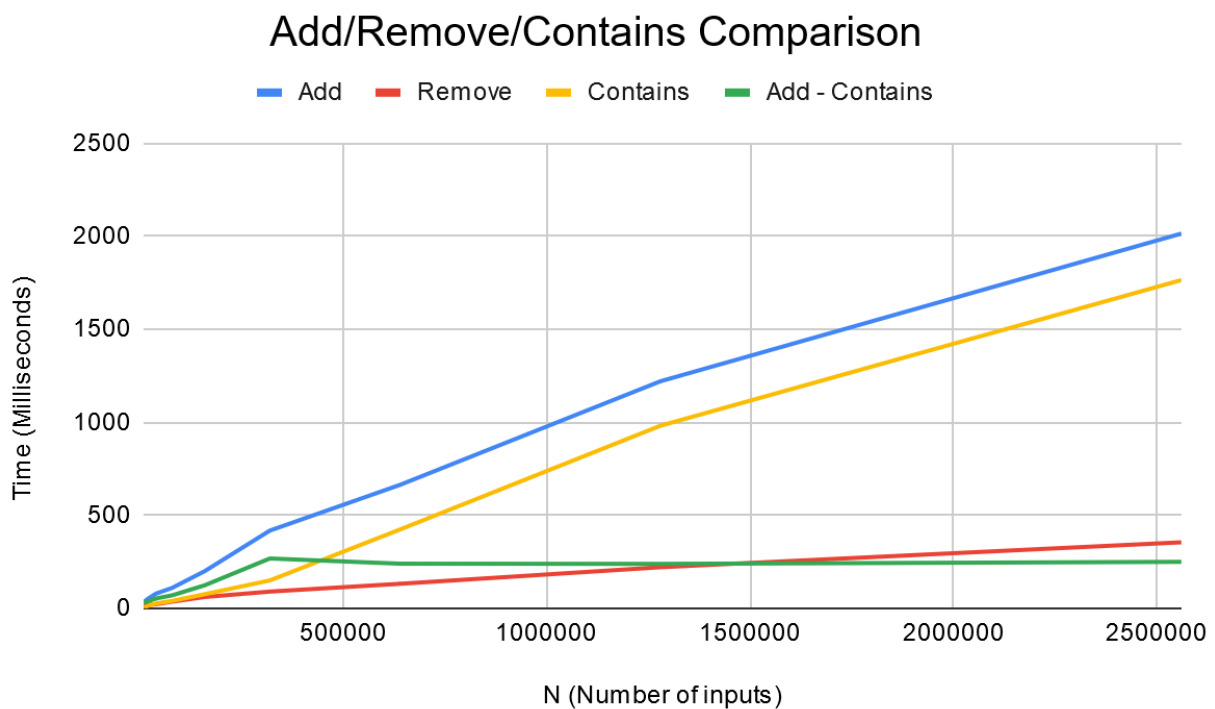
Because of the nature of the binary search tree we can expect many of the operations to take $O(\log(n))$ time to complete. Specifically the add, remove, and search (in this case contains). For the union, intersection, and difference methods, we will expect them to run in roughly $O(n)$ time, it will be longer by a constant because it will be performing operations in 3 different sets, those being comparing the contents of two array lists and then adding the appropriate values to a third set which will be done in $O(\log(n))$ time. Finally is the hashCode method and the equals method. For my custom hashCode I generate the default hashCode for each Item in the set and sum them together then multiply that sum by 13 and get the hashCode by taking the modulus of that against the size of the set. This will result in unique hashcodes for every single set unless the contents of each set is equal. The equal method will generate the hashCode for each set and compare them. Because these involve interacting with each element in the set we can expect this to scale linearly, however due to the speed of these operations it should scale very slowly.

An additional note about this process. All values in the charts and in the graphs were generated by taking the average of 10 runtimes at each size. In the graphs there are no trendline or trendline equations, when transferring the graphs from excel to this document the dotted trendlines and their associated equations would not show up no matter what I did. Instead at the end of each part I include a section discussing the trendlines and give there equations with their associated R^2 values to mediate this and help illustrate the power the trendlines have with their respective plots.

Part 2

Comparison between Add, Remove and Contains methods

n	Add	Remove	Contains	Add - Contains
10000	34	17	4	30
20000	51	18	13	38
40000	79	22	26	53
80000	110	36	40	70
160000	200	61	76	124
320000	418	90	150	268
640000	664	132	424	240
1280000	1221	221	982	239
2560000	2015	355	1765	250



Above is both the chart and the graph comparing the Add, remove, and contains methods from my set program. I chose to scale our number of inputs by doubling the number of inputs after each test because this has been a standard in our class along with it making it easier for us to tell how each of these methods scale. For example, if a method performs linearly, we expect that if the number of inputs doubles, then the runtime of the method should also double. If it is less than double, then it is performing sub linearly and scales faster than linearly if it is above double the previous runtime. All three of these methods performed sub linearly then by that idea, the runtimes never were double or

above their previous run times when the number of inputs doubled. One interesting thing to note was, do to the nature of a set and a binary search tree, we cannot allow duplicates and therefore much check if item to be added is already in the tree. I did this by calling the contains() method for every addition. After looking at the results I saw that the trends for contains and add were visually very similar so I added a plot of the add runtime minus the contains run time which you can see in green. It appears that the majority of the run time for the add method was caused by the interval contains check. This made it appear that without this check the runtime for the add method was much closer to that of the remove method which is what we should expect. All of these methods perform between logarithmically and linearly with the remove and add – contains being more logarithmic in nature and the add and contains being closer to linear. The logarithmic equations for these equations are as follows

Add : $310\ln(x) - 3188$ with an R^2 value of 0.756

Remove: $53.7\ln(x) - 538$ with an R^2 value of 0.787

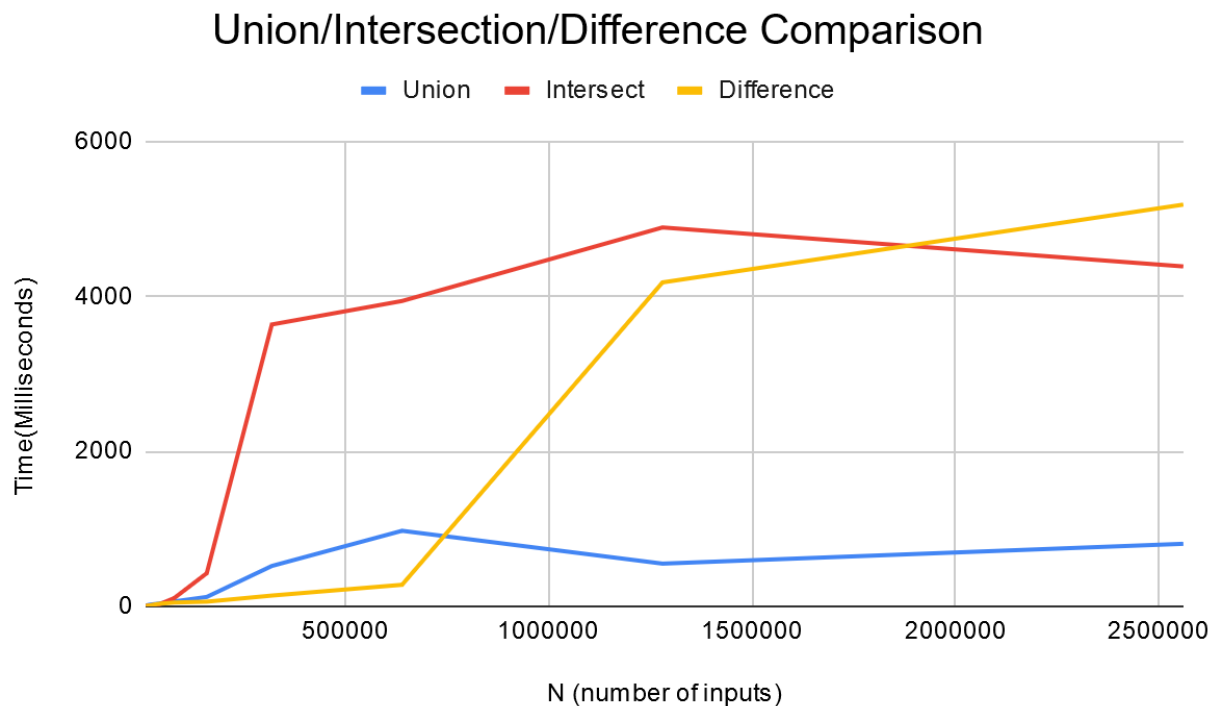
Contains: $261\ln(x) - 2742$ with an R^2 value of 0.716

Add – Contains: $49.4\ln(x) - 446$ with an R^2 value of 0.846

Part 3

Comparison between Union, Intersect and Difference methods

n	Union	Intersect	Difference
10000	9	3	4
20000	21	12	11
40000	34	19	31
80000	63	106	49
160000	121	427	60
320000	521	3637	139
640000	975	3940	278
1280000	551	4889	4180
2560000	807	4386	5184



The above chart and graphs shows the performance of our three comparison methods for my set class. These three performed very strangely considering we expected them to be at least somewhat linear in nature. During testing there were certain times when the runtime for one of these three spiked and the others dipped and vis versa on other run throughs. This was most likely caused by just the randomness of data that filled these sets that had a drastic effect on the runtime with the way they were implemented. For the union method I simply added all values of the first set to a new third set then attempted to add every element of the second set to it, because the add method would catch and not add duplicate values. For the intersection method I used a for each loop to iterate through one set while comparing each value to the contents of the other and added to the third set if the item was shared in both sets. The difference method functioned the same way but would only add if the item was found in one set and not the other. The data does not show a clear trend for these methods and they were inconsistent across run times but with these averages and experimenting with trendlines I was able to find that the trends with the highest R^2 value was with the logarithmic trendline. There equations are as follows:

Union: $171\ln(x) - 1707$ with an R^2 value of 0.749

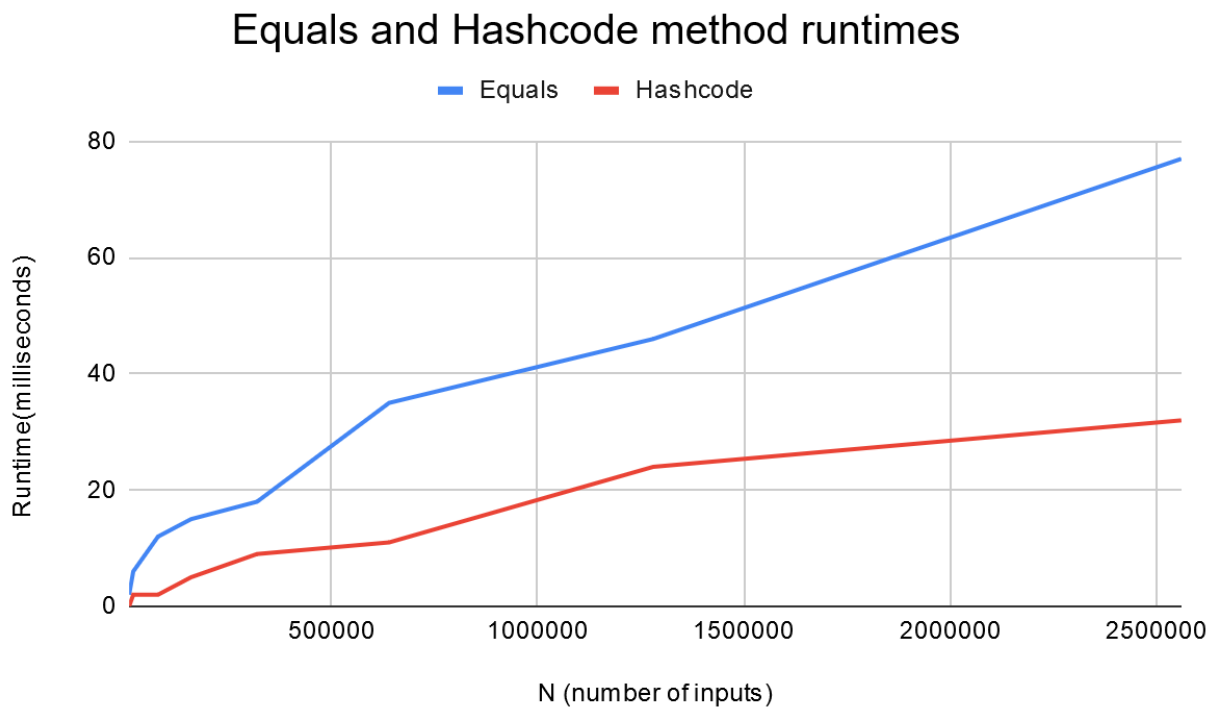
Intersect: $1047\ln(x) - 10609$ with an R^2 value of 0.823

Difference: $813\ln(x) - 8638$ with an R^2 value of 0.721

Part 4

Discussion of hashcode and equals methods

n	Hashcode	Equals
10000	0	2
20000	2	6
40000	2	8
80000	2	12
160000	5	15
320000	9	18
640000	11	35
1280000	24	46
2560000	32	77



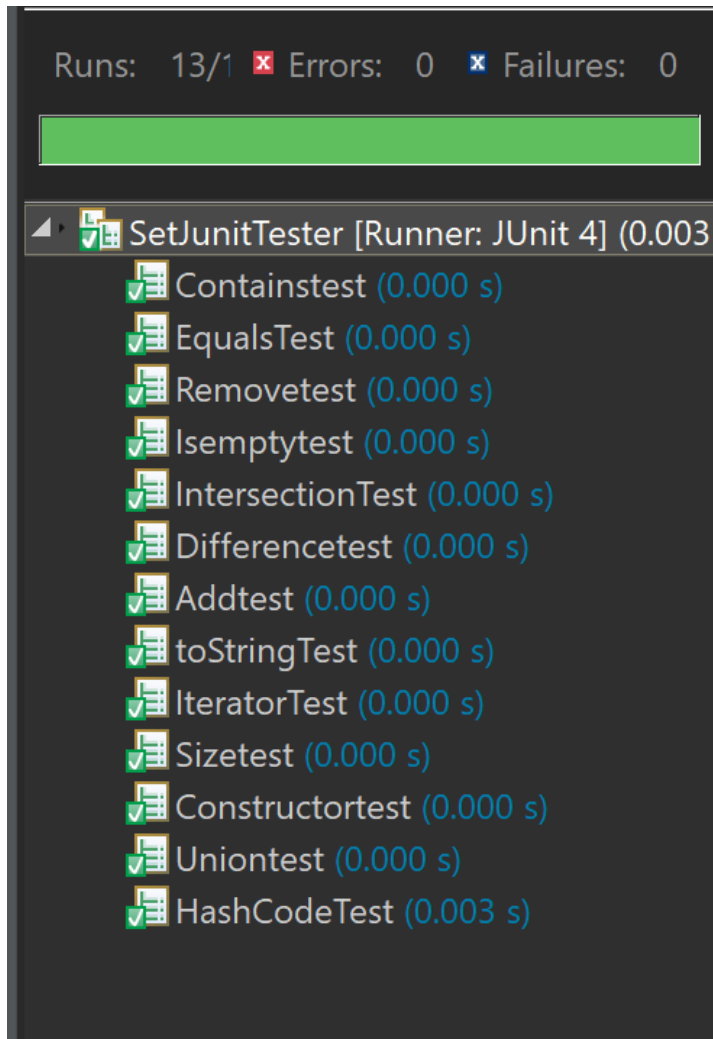
Above are the chart and graph showing the runtimes of the hashcode and equals methods runtime throughout the experiment. As expected they both seem to be somewhat linear in nature and the equals method had about double the runtime of the hashcode method which makes sense as the equals

method consisted mainly of running the hashCode method twice. The trendline equations for these plots along with associated R^2 values are as follows:

HashCode: $1.27e^{-5}x + 2.44$ with an R^2 value of 0.946


Equals: $2.8e^{-5}x + 8.42$ with an R^2 value of 0.968

JUnit Tests



The above 13 tests were to test the basic functionality of all of the 13 main methods of this program. There are additional helper methods in the program that while not directly tested were used in the testing of these 13 methods of focus. The set was tested and experimented with Integers. All of these tests consisted of testing the methods thoroughly as to cover the whole range of the programs capabilities.

Coverage:

SetJUnitTester (May 3, 2020 2:36:27 PM)				
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
SE374_Program6	 88.5 %	1,208	157	1,365

Our coverage value from our Junit tests was 88.5%. This is lower than we would ideally want but the majority of the lines that were not reached were the lines in the other tester, stdrandom and the stopwatch class that were not tested in our Junit tests. Besides this the vast majority of the set class was tested with few exceptions in conditionals where some lines of code were never met.