12/2/2016

Project 5: Hash Tables

CS249

Instructor: Patrick Kelley

Stephen White

Overview: The purpose of this project was to create two separate data structures that would model a hash map and a hash set respectively, to profile those data structures, and to create an application that would utilize a hash set to determine whether or not certain words were actual words according to the wordlist.txt that was provided to us.

Process:

Hash Set: To program a hash set, I simply implemented the ISet interface provided to us, and thought about how I was going to create a completely generic array. This took all of ten minutes to figure out, and then I went to work on the add method. The main thing with add for my hash set was that it utilized double hashing to handle collisions. What this meant in code form was that once I obtained the hash code of the element and utilized our hash function to obtain the index at which it would placed, if a collision occurred, I would simply convert the previously obtained hash code into a string, generate that string's hash code, and re-calculate the index using the same function (Math.floorMod()). With that accomplished I incremented the size by one. The has() method was very similar to the add method, except for the fact that it utilized the .equals() method to determine if two generic objects were the same or not. In the event that the calculated index did not return a null pointer, the same type of double hashing would occur and I would check to see if I had the correct element in my grasp. If it was, I would return true, but in the event that I ended up with a null pointer, I would return false. My size method would simply return the instance variable size that was incremented by one every time I added an item to the hash array, and my getInternalArray() method would just return my hash array as is. In terms of the toString(), I created a temp variable that would reference the number of items that had been added to the array, and would decrement every time I wrote out that item's toString(). This would, in turn, allow me to determine whether or not I needed to add a comma and a space to my string. Once the temp size became 0, I would add the last element, and close off the string.

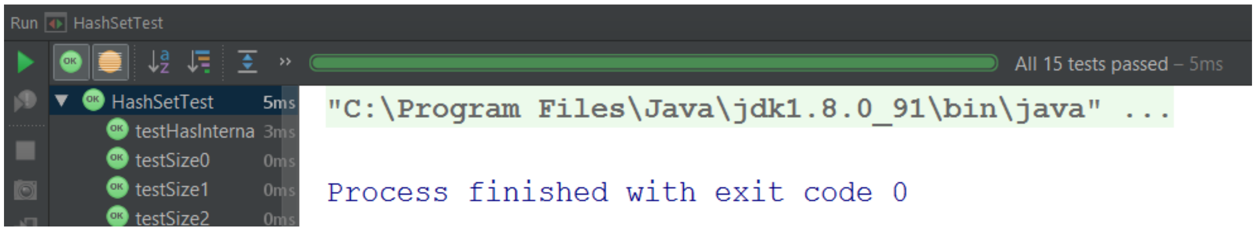
Hash Map: Programming a hash map was essentially the same as programming a hash map, just with slight variations here and there. The one major thing that made this portion of the project differ from what I had just done, was the fact that I had to utilize separate chaining to handle collisions. Right off the bat I knew that this meant that I had to create an inner Node class. This inner class would implement IMap.IMapPair, which was an inner interface that the original IMap interface contained. This node class was similar to every other node class we have written in the semester: it contained a key, an element, and a reference to a next node. I created to a constructor for the Node class that took in a key as well as an element. The node class had methods to get its element, get its key, set the next node in the sequence, and to obtain the next node in the sequence. With my node class out of the way, I began working on my put() method. It would generate a hashed index in the exact same way as my hash set, however it would utilize the key to create the hash code instead of the element directly. At this point, if there were any collisions here is what I would do to handle them: They item to be inserted was placed inside of a node (key and element), and if there was a collision, I would travel along that chain's links until I reached the end (a null pointer) and wire the new node in place. Get worked exactly the same, except in the event that I hit that null pointer and I had not found an element based on its key, I would throw a NoSuchElementException. My size and getInternalArray() methods were the exact same as my hash set, and my toString() went through the same steps as my hash set, except it printed items out in the form "Key: element," by iterating through the hash array and traveling along all links in the process. Check the results section to see screenshots of all unit tests passing.

Hash Map & Hash Set Profilers: The directions were absolutely horrible at describing what I was required to do, so I figured I would describe my thought process here. Both profilers worked in the exact same way, as they created 3 arrays of size 300,110, and filled them with 100,000, 200,000, and 300,000 items respectively. From there, I timed how long it took to find the 100,000, 200,000, and 300,000 items ten times each. If you take a look below at the results, you will see the following trend: the hash map is approximately 3 million nanoseconds (3 milliseconds) slower than the hash set. This has to deal with the fact that once we obtain an index, we must perform a linear search to find what it is that we are looking for in the sequence of nodes. In a hash set, we must simply perform calculations to jump to the next index, therefore shortening the amount of time taken to locate any given item.

Hash Set Scrabble Application: This small application was no big deal to create. I wrote some code that would read in *wordlist.txt* line by line and throw each line into my hash set. After I had accomplished this, all I had to do was use my use() method to determine if certain words were actual words. See results section for a screenshot.

Results:

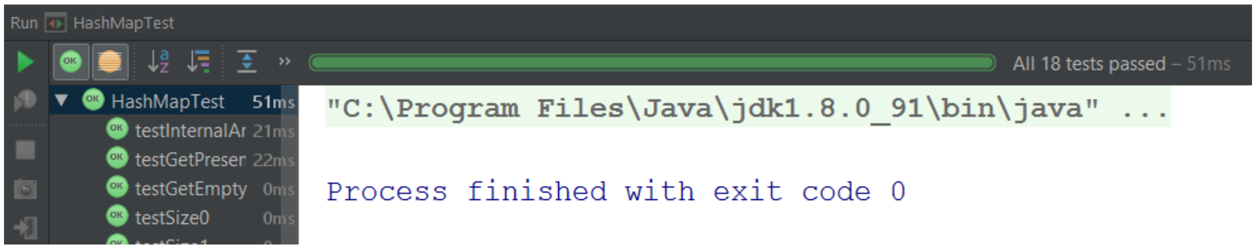
All Unit Tests Passing:



```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...

Process finished with exit code 0
```

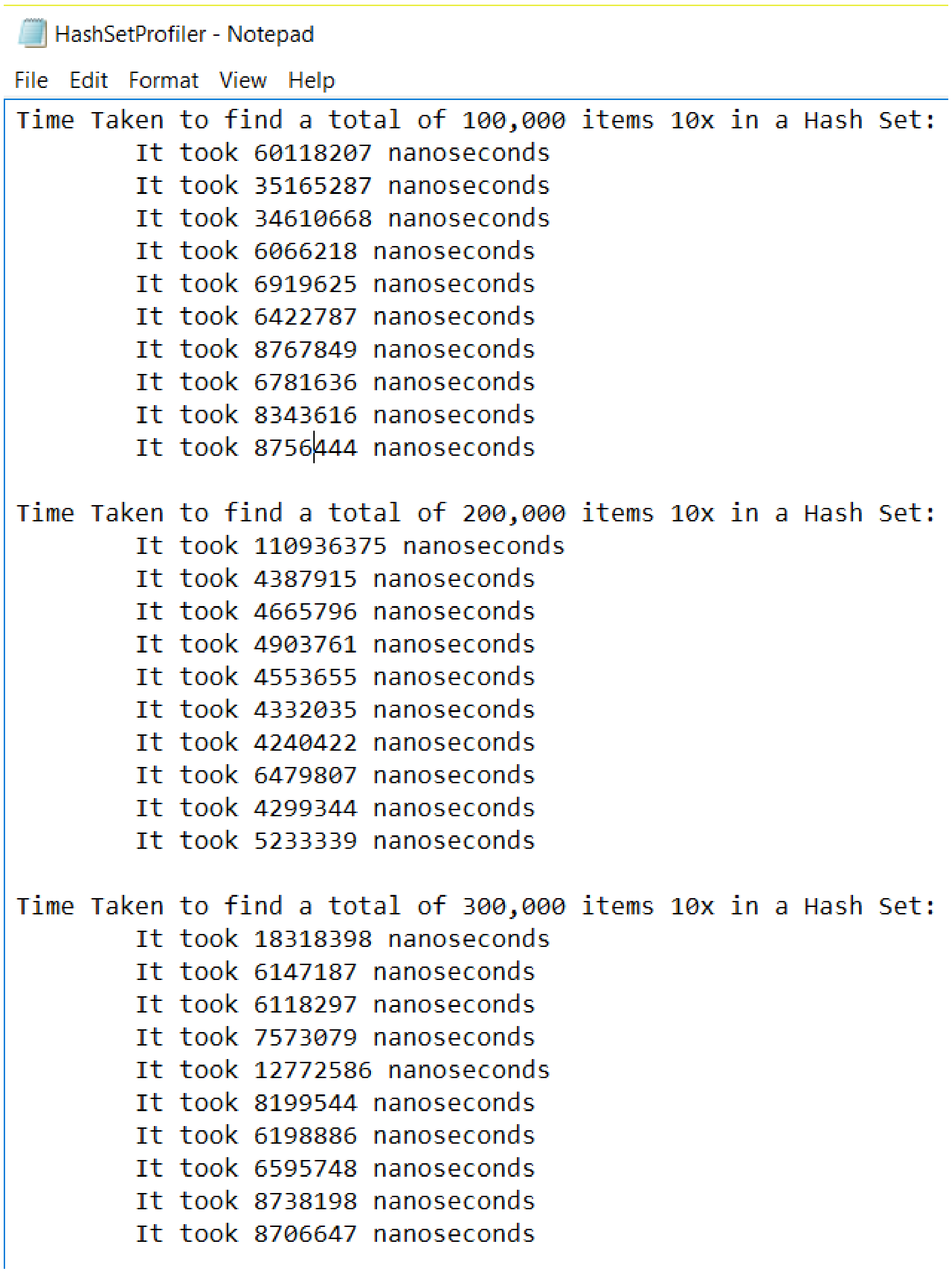
Hash Set Profiler:



All 18 tests passed – 51ms

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...

Process finished with exit code 0
```

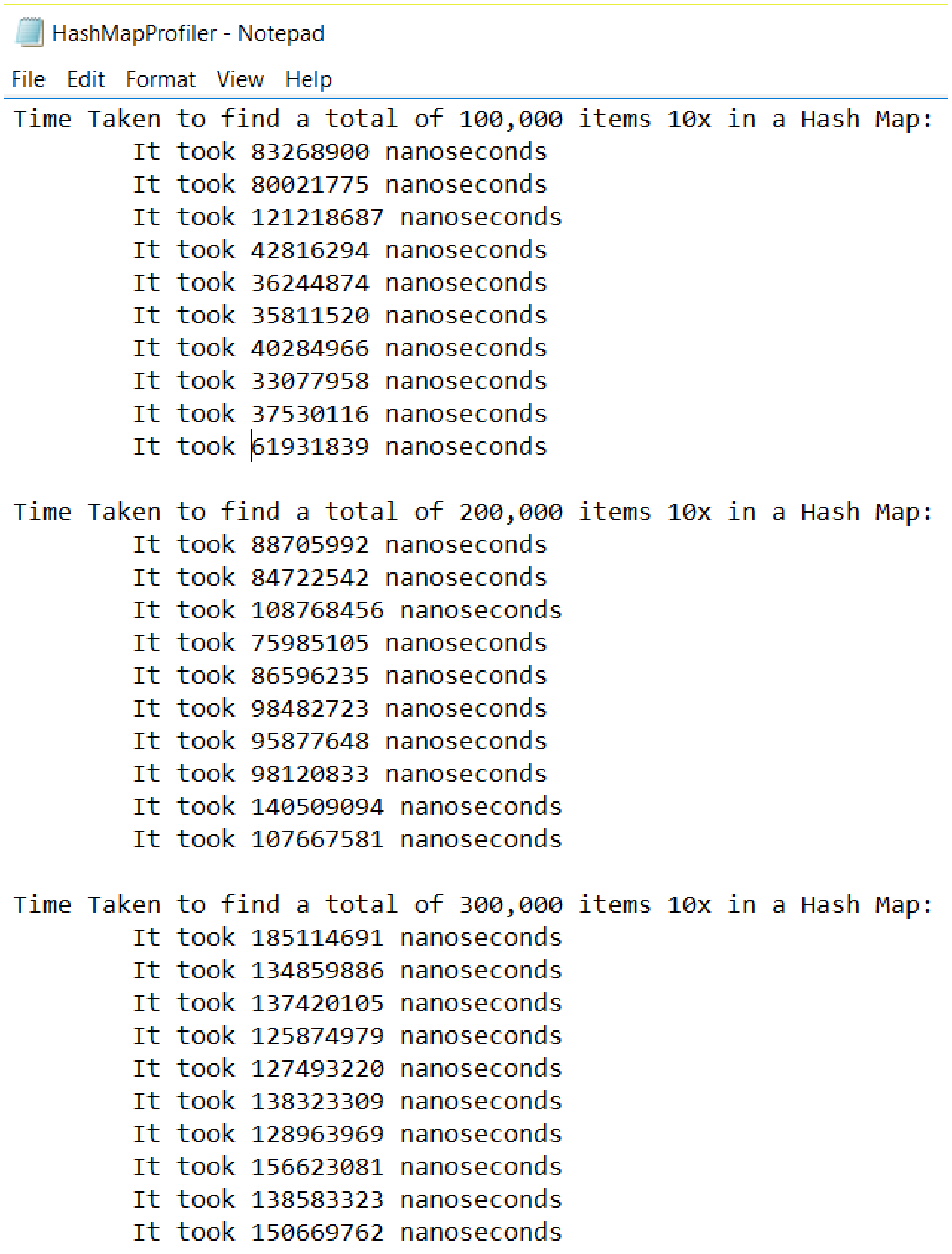


```
Time Taken to find a total of 100,000 items 10x in a Hash Set:
        It took 60118207 nanoseconds
        It took 35165287 nanoseconds
        It took 34610668 nanoseconds
        It took 6066218 nanoseconds
        It took 6919625 nanoseconds
        It took 6422787 nanoseconds
        It took 8767849 nanoseconds
        It took 6781636 nanoseconds
        It took 8343616 nanoseconds
        It took 8756444 nanoseconds

Time Taken to find a total of 200,000 items 10x in a Hash Set:
        It took 110936375 nanoseconds
        It took 4387915 nanoseconds
        It took 4665796 nanoseconds
        It took 4903761 nanoseconds
        It took 4553655 nanoseconds
        It took 4332035 nanoseconds
        It took 4240422 nanoseconds
        It took 6479807 nanoseconds
        It took 4299344 nanoseconds
        It took 5233339 nanoseconds

Time Taken to find a total of 300,000 items 10x in a Hash Set:
        It took 18318398 nanoseconds
        It took 6147187 nanoseconds
        It took 6118297 nanoseconds
        It took 7573079 nanoseconds
        It took 12772586 nanoseconds
        It took 8199544 nanoseconds
        It took 6198886 nanoseconds
        It took 6595748 nanoseconds
        It took 8738198 nanoseconds
        It took 8706647 nanoseconds
```

Hash Map Profiler:



```
Time Taken to find a total of 100,000 items 10x in a Hash Map:
        It took 83268900 nanoseconds
        It took 80021775 nanoseconds
        It took 121218687 nanoseconds
        It took 42816294 nanoseconds
        It took 36244874 nanoseconds
        It took 35811520 nanoseconds
        It took 40284966 nanoseconds
        It took 33077958 nanoseconds
        It took 37530116 nanoseconds
        It took 61931839 nanoseconds

Time Taken to find a total of 200,000 items 10x in a Hash Map:
        It took 88705992 nanoseconds
        It took 84722542 nanoseconds
        It took 108768456 nanoseconds
        It took 75985105 nanoseconds
        It took 86596235 nanoseconds
        It took 98482723 nanoseconds
        It took 95877648 nanoseconds
        It took 98120833 nanoseconds
        It took 140509094 nanoseconds
        It took 107667581 nanoseconds

Time Taken to find a total of 300,000 items 10x in a Hash Map:
        It took 185114691 nanoseconds
        It took 134859886 nanoseconds
        It took 137420105 nanoseconds
        It took 125874979 nanoseconds
        It took 127493220 nanoseconds
        It took 138323309 nanoseconds
        It took 128963969 nanoseconds
        It took 156623081 nanoseconds
        It took 138583323 nanoseconds
        It took 150669762 nanoseconds
```

Big O Analysis:

Hash Set:

Method	Big O Notation
Public HashSet(int size)	O(1) - every operation was constant
public void add(E element)	O(1) - required only basic math to calculate the proper index in the array
public boolean has(E element)	O(1) - required only basic math to calculate the proper index in the array and to determine whether or not the element was present
public int size()	O(1) - every operation was constant
public Object[] getInternalArray()	O(1) - every operation was constant
public String toString()	O(N) - we must iterate over (n elements - # null values) = N elements

Hash Map:

Method	Big O Notation
Public HashMap(int size)	O(1) – every operation was constant
public void put(E element)	O(N) – required at worst case to search through an N-link chain to place the node
public E get(K key)	O(N) – Required at worst case to search through an N-link chain to obtain a node
public int size()	O(1) – every operation was constant
public Object[] getInternalArray()	O(1) – every operation was constant
public String toString()	O(N) – we must iterate over (n elements - # null values) = N elements

Conclusion: This project was extremely easy compared to what I have become acclimated to. With the introduction of Red Black Trees, I absolutely lost faith in myself as a programmer for that short period of time, because it was such a challenge for me to wrap my head around, and even after numerous explanations, I still was not able to code that portion of the project on my own. With hash sets and hash maps, I was able to look up a couple YouTube videos, and after approximately half an hour, I would say that I was confident enough to start this project on my own. Not only did I start the project on my own, as I had every project beforehand (excluding RBTrees of course), but I was also able to complete the entire project without any outside assistance. I did utilize my peer Connor Scwhirian to help clean up some code after I had finished however, and I did teach a numerous amount of other people how the project worked, and was able to come out on top. With this being the last project of the semester, I can say with full confidence that I have given this course every single bit of strength I had, and never gave up until I saw that all of your basic unit tests had passed. If I have learned anything from this course, it would be that it is ok to fail, so long as you get back up and continue trying. To utilize every resource, ask every question, and exhaust every idea you have, and then some, before even thinking of quitting. Thank you for a challenging semester, one of which I will never forget, and will attribute much of my determination and success as a programmer.