Time Measurements

Student Marcus Johansson

Department of Computer Science, Linnaeus University
Magna floor 4, 391 82 Kalmar Sweden
mj223vn@student.lnu.se

Abstract - **The task handled in this report is to find out how many concatenations can be done in one second . We use two methods for concatenation. 1) Using the** str += "..." **and 2) Using StringBuilder class and method** append(..) . **We use the methods on short strings adding strings containing only one character and long strings adding strings representing a row with 80 characters. We are interested in the number of concatenations and the length of the string when using the two methods. Our experiment shows that on shorts strings the** str+= **method can do about 100000 concatenations and using StringBuilder** append( ) **method we get about 310 million appends in one second.**

I. Exercises

The problems we handle in this report is the following: a) How many short strings concatenations can be done in one second using the str+= method. b) How many short strings concatenations can be done done in one second using StringBuilder method append( ). c) How many long string concatenations can be done in one second using the str+= method. d) How many long strings concatenations can be done using StringBuilder method append( ). The length of the string will be the same as number of concatenations in the short string experiments and concatenations times 80 in the long string experiments.

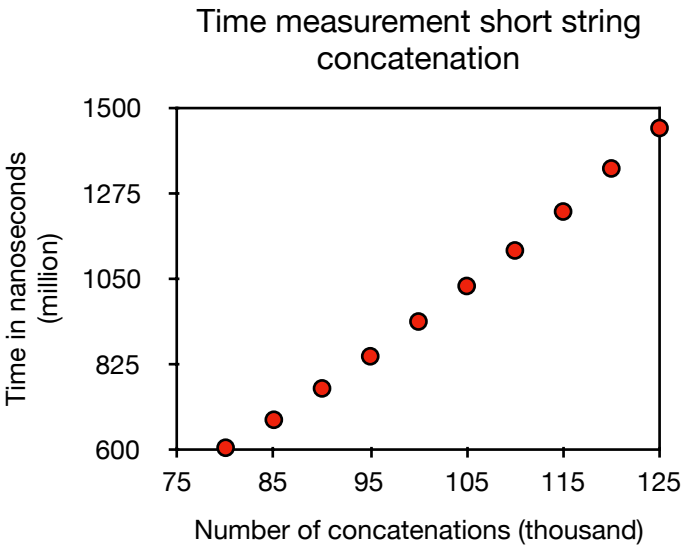
II. Experimental setup

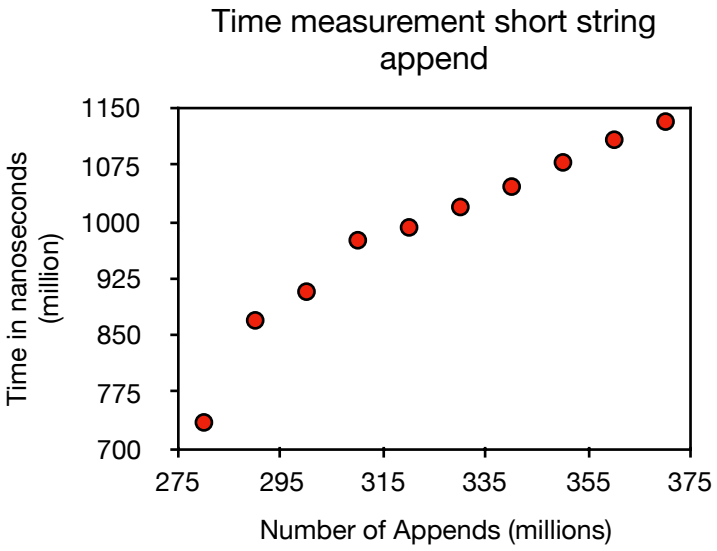
All experiments was done on a MacBook Pro with an Intel core i5 processor 2 GHz with 8GB of memory. We used Java 8 201. To get as accurate measurements as possible we closed all other applications and nothing else then then concatenation running in the code during the time measurement. We are using the same approach in all four measurements. An garbage collection is run before each test to to avoid having memory problems. We start with five warm up runs of the program to optimizing the JVM for our code. Then we measure the time for the concatenations ten times and get the average time by dividing with ten. We use our own method timeNumOfConcatenations(**int** size, **int** testRuns, **int** warmUps). The input size is number of concatenations to be executed, testRuns is the number of times to run the test and warmUps is the number of warm ups to be

```
public static void main(String[] args){
  int numOfApp = 280000000;
  int tests = 10;
  for (int i = 0; i < tests; i++){
    System.out.print("Numbers of appends = " + numOfApp + ", ");
    timeNumOfConcatenations(numOfApp, 10, 2);
    numOfApp = numOfApp + 10000000;
   }
}
```

To find the number of concatenations close to one second we used an try and error approach. We did the same for all four experiments. For example if we use StringBuilder we can concatenate around 310 000 000 in one second. We then use an interval [280, 350] millions to to cover the number of concatenations we are looking for. A print out when running the code above look like this:

```
Numbers of appends = 280000000
Elapsed time: 735039644 nanoseconds
String length = 280000000
Memory = 571MB
```





Short string concatenation			
Number of concatenations	**String length**	**Time (nanoseconds) (million)**	**Memory (MB)**
80000	80000	603	120
85000	85000	677	302
90000	90000	760	263
95000	95000	845	157
100000	100000	937	243
105000	105000	1031	7
110000	110000	1125	163
115000	115000	1228	20
120000	120000	1342	89
1250000	125000	1449	25

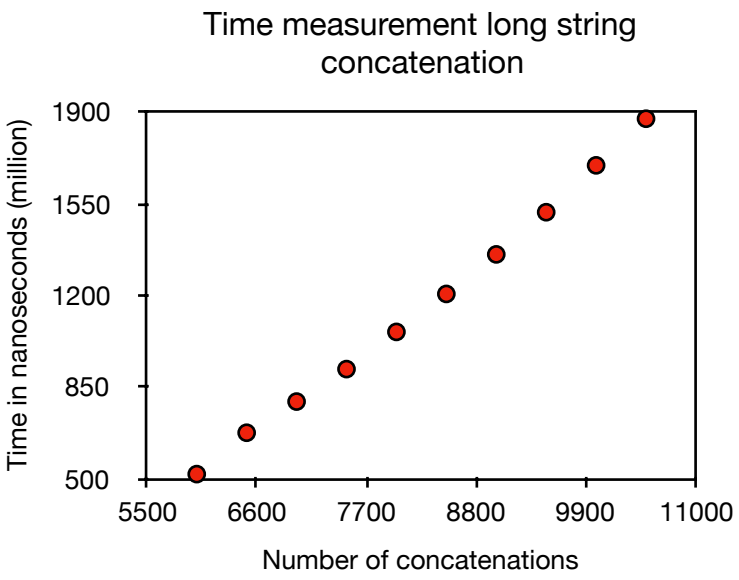
Short string StringBuilder			
Number of appends (million)	**String length (million)**	**Time (nanoseconds) (million)**	**Memory (MB)**
280	280	735	571
290	290	870	455
300	300	908	455
310	310	976	908
320	320	993	908
330	330	1020	908
340	340	1047	908
350	350	1079	908
360	360	1109	908
370	370	1066	908

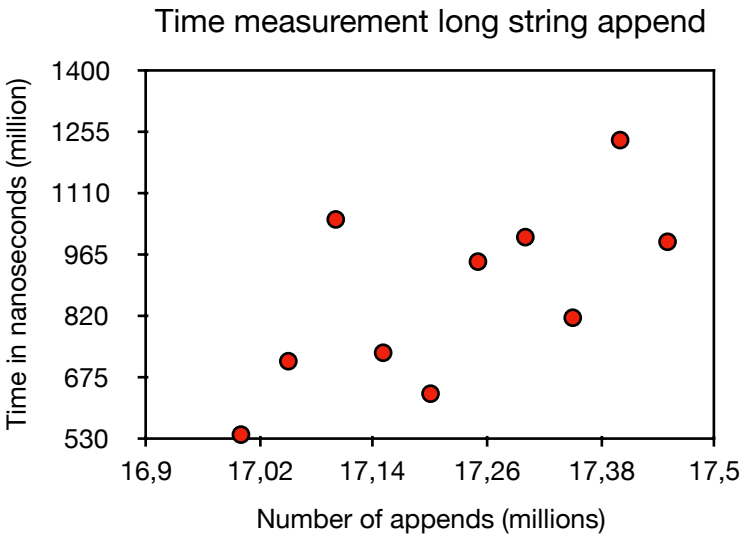
III. Short strings

The tables and diagrams above show how the different measurements went by. The first thing to observe is that the number of concatenations using stringBuilder is significantly larger and the memory use is also much larger.

The diagram for the appends is not increasing in a straight line. This might be a flaw in our warm up technic.

Form the tables we can se that the += approach can handle about 100000 concatenations in one second while StringBuilder can handle about 310 000 000 concatenations in one second.





Long string concatenation			
Number of concatenations	**String length (million)**	**Time (nanoseconds) (million)**	**Memory (MB)**
6000	480000	517	309
6500	520000	675	432
7000	560000	794	215
7500	576000	918	43
8000	584000	1060	272
8500	600000	1205	269
9000	616000	1356	273
9500	624000	1517	30
10000	640000	1696	290
10500	680000	1874	331

Long string StringBuilder			
Number of appends (million)	**String length (million)**	**Time (nanoseconds) (million)**	**Memory (MB)**
17	1360	537	2066
17,05	1364	711	2737
17,1	1368	1047	2737
17,15	1372	731	4214
17,2	1376	634	3525
17,25	1380	947	3525
17,30	1384	1005	3525
17,35	1388	814	3525
17,4	1392	1235	3525
17,45	1396	994	3525

IV. Long strings

The tables and diagrams above show how the different measurements went by. The first thing to observe is that the number of concatenations using stringBuilder is significantly larger and the memory use is also much larger.

In the measurements for StringBuilder we can se that it's not a steady graph. That is a problem we do not fully understand. Several test runs was made with similar results. To grasp that behavior we need to investigate further, but that will not be included in this report. Also the memory usage is much larger then using str+= method.

V. Conclusion

As we can see from the test results, using StringBuilder to do concatenations is a much more effective approach. The reason for this is that when using the str+= method we create a new String object and replacing the reference with the new string. Each new concatenation requires the construction of an entirely new String object.

StringBuilder use an internal array and simply updates the array when changes are made. This means that new memory is only allocated when the array needs to be resized.

To improve our test further we can use regression to calculate an more precise position of the one second point. But for this experience we say its "good enough" to do an estimation based on the graphs on where the one second point is.