CSE 100: PA2 Final Report

Robin Heinonen

October 30, 2017

1 Dictionary benchmarking

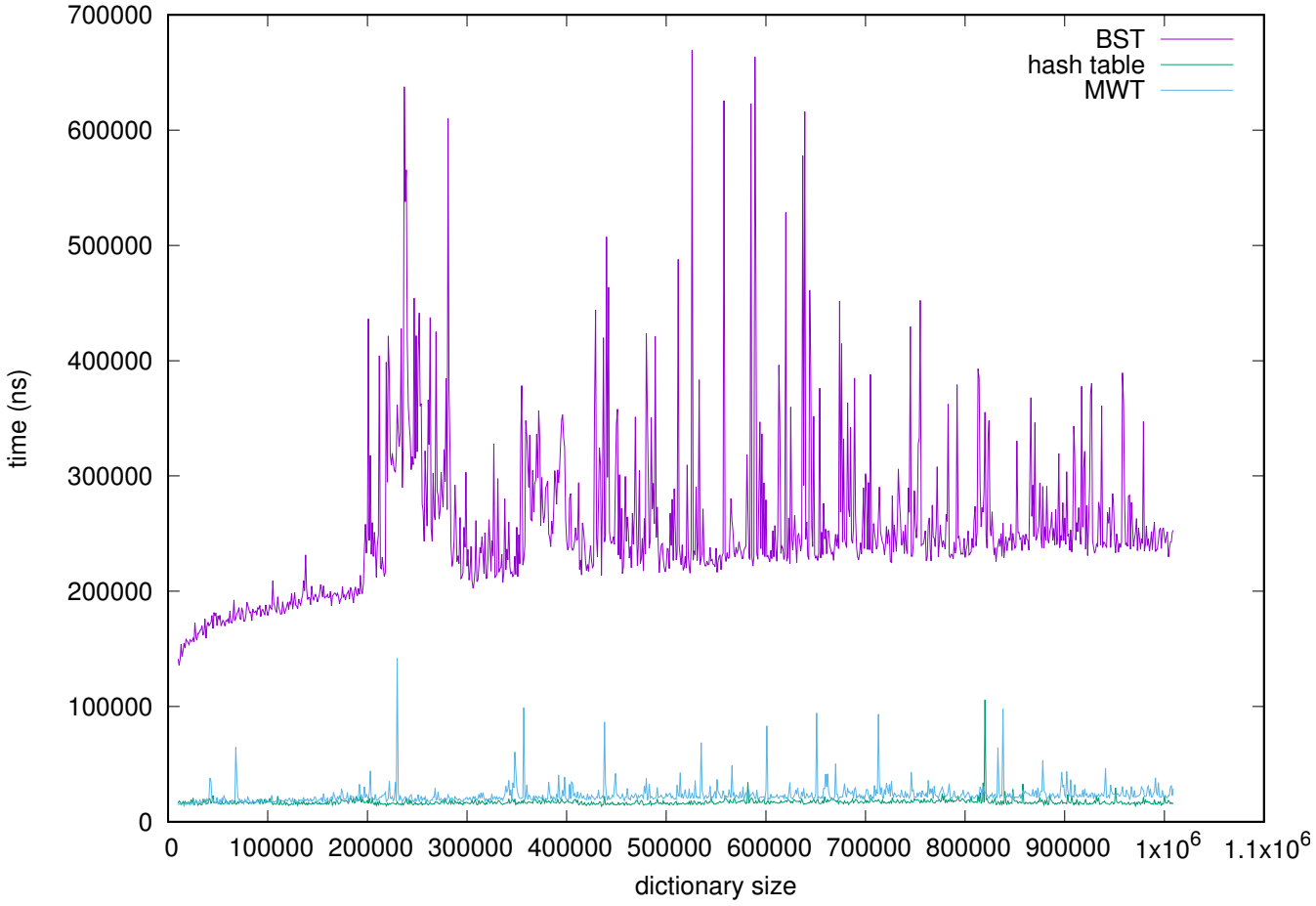

Figure 1: Plot showing the runtimes for finding 100 elements not in the dictionary, for the three implementations.

Fig. (1) shows the runtimes in nanoseconds for finding 100 words (not in the dictionary) in my three implementations of the dictionary (a binary search tree, a hashtable, and a multiway trie). The results are generally consistent with expectations. The curve for BST

appears to be logarithmic—it is certainly sublinear but definitely nonconstant—and indeed we expect a $O(\log N)$ worst and average case for (balanced) BSTs. The hashtable and MWT are much faster (with the hashtable a little faster than the MWT): both appear to be approximately constant. This again, is consistent with expectations, as the average case for the hashtable is $O(1)$ and the average and worst cases for a MWT are $O(k)$ (where k is the word length), and we don't expect word lengths to vary much.

It is worth mentioning the fluctuations present in these plots. They do not contradict the expected runtimes and can be attributed to a number of factors. For one, this benchmarking was done on my laptop while working, and computational speeds could fluctuate depending on the activity of background processes. This is probably the most significant effect, especially for the more intensive task of searching the BST. As already mentioned, the word lengths may change (causing the MWT timings to fluctuate), although this is a small effect. Moreover, varying numbers of collisions for different words will lead to fluctuations in the timings for the hashtable.

2 Hash function benchmarking

The two hash functions I benchmarked are the ELF hash and the DJB hash, both found at http://www.partow.net/programming/hashfunctions/.

The ELF hash works as follows:

1. Left shift the hash value (initially zero) 4 places bitwise and add the ASCII value of the next character in the string
2. If the leftmost 4 binary digits of the hash value are not all zero (i.e. (hash AND 0xF0000000)≠0), bitwise shift right the result of (hash AND 0xF0000000) by 24 places, and set hash to be the result of hash XOR this last quantity.
3. Set the hash value to the one's complement of the result of hash AND (the previous value of hash AND 0xF0000000)
4. repeat for each character in string

The DJB hash works as follows:

1. Start with the initial value of 5381
2. Add the current hash value to the value left shifted 5 bits
3. Add the ASCII value of the next character in the string
4. Repeat steps 2 and 3

Let's run some simple test cases: the strings "a," "it," and "dog."

2.1 ELF(a)

1. ASCII value of a is 97, or 0000 0000 0000 0000 0000 0000 0110 0001
2. above AND 0xF0000000=0
3. hash value is thus simply 97

2.2 ELF(it)

1. ASCII value of i is 105, or 0000 0000 0000 0000 0000 0000 0110 1001
2. above AND 0xF0000000=0
3. result of first iteration: 105
4. left shift four bits: 0000 0000 0000 0000 0000 0110 1001 0000=1680
5. add t=116: 1796
6. leftmost bits again zero, so hash value is 1796

2.3 ELF(dog)

1. ASCII value of d is 100, or 0000 0000 0000 0000 0000 0000 0110 0100
2. above AND 0xF0000000=0
3. result of first iteration: 100
4. left shift four bits: 0000 0000 0000 0000 0000 0110 0100 0000=1600
5. add o=111: 1711
6. leftmost bits again zero, so result of second iteration is 1711
7. left shift four bits: 0000 0000 0000 0000 0110 1010 1111 0000=27376
8. add g=103: 27479
9. hash value is 27479

2.4 DJB(a)

1. 5381<< 5 = 10101000010100000=172192
2. 172192+5381=177573
3. a=97
4. 177573+97=177670
5. hash value is 177670

2.5 DJB(it)

1. i=105
2. 177573+105=177678=101011011000001110
3. left shift five bits: 0000 0000 0101 0110 1100 0001 1100 0000=5685696
4. 5685696+177678=5863374
5. t=116
6. 5863374+116=5863490
7. hash value is 5863490

2.6 DJB(dog)

1. d=100
2. 177573+100=177673=101011011000001001
3. left shift five bits: 0000 0000 0101 0110 1100 0001 0010 0000=5685536
4. 5685536+177673=5863209
5. o=111
6. 5863209+111=5863320=0000 0000 0101 1001 0111 0111 1001 1000
7. left shift five bits: 0000 1011 0010 1110 1111 0011 0000 0000=187626240
8. 187626240+5863320=193489560
9. g=103
10. 193489560+103=193489663

All of these values are confirmed by the program.

2.7 Data

# of hits	# of slots (ELF)	# of slots (DJB)
0	127	124
1	52	53
2	15	6
3	6	1

Table 1: Collision data for a hash table of size 200

DJB consistently performs slightly better than ELF in that the average and worse case search times are lesser. DJB, therefore, is the better hash function.

# of hits	# of slots (ELF)	# of slots (DJB)
0	1329	1218
1	457	593
2	136	162
3	54	26
4	15	0
5	6	1
6	2	0
7	1	0

Table 2: Collision data for a hash table of size 2000

# of hits	# of slots (ELF)	# of slots (DJB)
0	13242	12124
1	4538	6047
2	1493	1571
3	514	223
4	153	33
5	45	2
6	8	0
7	7	0

Table 3: Collision data for a hash table of size 20000

table size	average steps (ELF)	average steps (DJB)	worst case steps (ELF)	worst case steps (DJB)
200	1.33	1.25	3	3
2000	1.499	1.246	7	5
20000	1.467	1.2458	7	5

Table 4: Successful search time data