Ashley Nguyen - apn2my
PostLab06 - postlab6.pdf
3/17/16

Big Theta

The Big-Theta running time of my application is linear with the vector< list<string> > implementation. For insertions, the program read in the dictionary input file, reads each line of that file (containing a single word/string) and inserted it into a spot in the table one by one. To find the word (contains method) the program searched the list of strings from beginning to end (linearly), to see where the target word is and then utilizes the hash function to find its spot in the hash table to retrieve the key. Optimizations to improve big-theta involved modifying the hash function, which is the fundamental operation for insert and find (contains). The Big-Theta generalized equation solely for reading in input and searching is rows x columns x 8 based on the quad-nested for loop that ran for each search. (8 as the direction and max value in the third part of the loop).

The worst-case time complexity is: rows x columns x 8 + words where words is constant

Timing Information

Worse hash function → Str[0] % table size

Time = 6.219 seconds (words2.txt & 300x300)

Time = 55.2882 seconds (words.txt & 250x250)

The performance was worse because by only looking at the first character of the string and using it to assign a spot in the hashtable can cause many collisions, as many words in a dictionary have the same first character (ex. Apple, aardvark, etc.). This costs running time performance as the

application will have to search through the chained spot in the dictionary to retrieve the found word in the grid file.

Worse hash table size → when size = 0,

tableSize += getNextPrime(size)*getNextPrime(size)

Time = 31.4289 seconds (words2.txt & 300x300)

Time = 641.347 seconds (words.txt & 250x250)

This worsened the performance was due to creating an arbitrarily and extensively large hash table size that may not be entirely used. This inefficient table size calculation created an exponential number of spots in the hash table. When dealing with collisions via linear probing, the word needed to be inserted will be hashed to the next empty spot in the table. When using the find operation on this large table, it will linearly search through each spot until the word is found. Since there are more spots, there are more "steps" to search thus causing the performance to take longer.

Optimizations

*Run on Mac OS in VirtualBox

	Original time = 5.9774 sec
Optimizations	**Words2.txt & 300x300** **OPTIMIZED**
1. Modified return value of hash function to unsigned int	3.44 sec
2. Incremented hashSpot in hash function	3.365 sec
3. Hashspot %= capacity + 1	3.21 sec
4. Return hashSpot + capacity + 1	3.22 sec

5. Constructor Updated size to getNextPrime(size) * 2	3.10 sec
6. Updated Hash function str[i] * 2 % tableSize	3.26
7. Updated Hash function Hash value *= str[i] * 3 % tableSize	2.965

Overall speedup - words2.txt & 300x300
5.9774 / 2.965 = 2.016

Optimization descriptions

1. Returning an unsigned int for the hash function ensured that the spot the value is being hashed to will always be positive, thus allowing the program to ignore negative integers.
2. Although incrementing hash spot value by 1 before performing the hash function reduced the time by nearly 0.1 seconds, this optimizes the application by pushing the spot by a constant value to reduce collisions and improve run time.
3. By modding the hash value by the maximum table size + 1 ensures that the key is being hashed to the very last spot in the table. Optimizing the hash table and the number of keys it can hold goes a long way to improve performance by determinism.
4. If the mod happens to be an integer less than 0, adding the hash value to the capacity + 1 seemed to improve run time by dealing with collisions slightly better by double checking that a key will not be accidentally hashed to a nonexistent spot in the table (the + 1).
5. Doubling the size of the table improves performance time by dealing with collisions in a better way through linear chaining. Ensuring the hash table size is twice as large as the keys desired to input avoids segmentation faults and un-hashed keys (providing faulty input).

6. In attempting to improve the hash function by multiplying the string's total characters by the smallest prime number did not work efficiently since the prime number used was 2 and could produce collisions based on the nature of this number.
7. I updated the hash function to use multiple the hash value by itself and multiply is by the character of the string which is multiplied by 3, the next highest prime number that is less than the previous prime number used 37 modded by the table size. This hash function had the fastest running time results thus far.

Other small optimizations/modification attempts:

Many unneeded variables and loops were omitted due to redundancy (such as directly using the method value, str.length() rather than using extra memory to store it in a variable). I tried to rewrite the lab using a vector<string> however, the run time was far worse than my original implementation of a vector<list<string>> so I maintained the latter to further optimize.