/*
 *Joseph Elsisi
 * lab 108
 *October 24
 */

The big-theta running time for my implementation is theta(r * c). To find this I multiplied the variables r and c because their lengths were used as the number of iterations in two nested for loops. There are 8 directions used as another nested for loop and would bring the equation to theta(8 * r * c) and finally there is a 4th nested for loop for word length, which in our case was a constant of 23. Now the equation is theta((8 * 23) * (r * c)) In the worst case, find, which was inside the 4 nested loops, would be called. In the worst case all the words would have hashed to the same linked list in my implementation. This would mean w calls each time because of linear find in a linked list, with w being the number of words in the dictionary. Another factor to consider is how many times a word is found. This is dependent on how many words there are in the grid, we can call this f. Now the big - theta is theta((8 * 23) * (r * c) + (fw). Because 23 and 8 are constants, we drop them. Since we only care about the highest ordered term, fn is also dropped. These values do not help describe the long term behavior of my implementation. I wrote a rehash function because of the other implementations I tried, but it will rarely be called and doesn't apply to the big-theta, just like when a vector has to be resized.

I ran my pre-lab implementation on my laptop 3 times for word2s.txt 300x300.grid.txt and got an average of 6.62499 seconds using the -O2 flag. Also note that this was done in the virtual box. The first change I made was to buffer the output. This cut my time significantly to an average out of three of 3.881596667. This means the speed up was 1.70677. I then tried linear probing and was surprised to find that it was slower, with an average time out of 3 of 4.284876667. I then tried quadratic probing and out of 3 got a similar time of 4.15029. FInally I

tried different powers for my hashing function but it made no difference. We learned in class that the power function in std math was very expensive, so in all my implementations I calculated the powers by hand, them in an array to save time.

Initially I picked a size about twice the size of the dictionary for my hash table. This was because I knew roughly what size the dictionary would be. I then picked a new size of 100, meaning that the rehash function would need to be called several times. Each time the load factor hit 0.75, the size of the table was doubled. When I ran this implementation, it took much longer. This is no surprise, as each time resize is called, a new array twice the size has to be made. Then, every element in the table has to be iterated through. Since each element is a linked list in my implementation, another list needs to be iterated through. This for loop inside a for loop means that the rehash function is theta(n^2) run time. All the elements in the table had to first be rehashed, aka finding a new hash value because of the new table size. Then the value is copied over to a new array, taking linear time. Although this is incredibly expensive, the rehash function does not to be called that often as long as a reasonably large table size is chosen from the start. All together, three runs of the small table size resulted in an execution time of 5.776 seconds. I was suprised to find that linear and quadratic probing would not run the fastest. We learned in class that using linked list buckets would cost more memory, and in the worst case, would have to iterate the entire linked list to find the value. This is worse than probing the entire table, because iterating through a linked list is much slower. This is in the worst case. My lab showed that the worst case does not accurately represent what an average run would take. On average, it is very rare for every word to land in the same bucket. So on average the separate chaining implementation wins.

Additionally, I mixed up my hash function out of three attempts, it took 4.173903 seconds. Instead of just multiplying the value of each char in the string by a power based on its

position. I took the first char in the string, and multiplied it by a current char being looked at in the for loop. Although not significant, the run time did increase. This method was designed to be poor because by using the first value as a basis, the hashes would overall not be evenly distributed.

Finally I tried a different container for my buckets, specifically the std set. I averaged a time of 9.14466. This is surprising as this should be the standard library implementation of a red-black tree, which guarantees log(n) for find, much faster than the linear time of a linked list. In the long run, this implementation could, and should, be faster than the linked list for separate chaining, but in this context and the chance that my implementation does not utilize its benefits properly, it was not able to speed up my program.