Ellis Saupe ems236
Final Report

1. **Description of data structures used**

I will analyze add, search, delete, and autocomplete methods of data structures designed to store words. The most useful application of this would be a simple spellcheck, so I will generally call the search method isWord. The data structures being used are an array, a tree, a hash table, and a variant of a hash table that stores additional information about substrings of valid words to be used for an autocomplete method.

The array stores whatever String is added at the end of the array. If it reaches its capacity, it doubles the size of the array. Searching linearly searches the array. The delete method searches for the word, removes it from the array, and shifts the rest of the array to the left. The autoComplete method searches the array for substrings that match the parameter and returns the first match.

The tree works as follows: The root of the tree contains no data. Each node of the tree contains an array of 26 references to its child nodes, and a boolean value signifying if this position on the tree is a valid word. To traverse the tree starting from the root, each character of the string corresponds to the index of the child – the String "ab" is found by starting at the root, setting the cursor to the a=0th child of the root, then the b=1st child of that node, and so on. The add word method traverses the tree using whatever word is being adding, initializes each node that is part of the transversal while setting the isWord boolean to false if necessary, and either initializes the node associated with the complete word if necessary and sets isWord to true. Every substring of a word must be initialized in the tree to access its children and store the word, therefore, a null value at some position in the tree signifies that no valid words contain the String associated with that position. The isWord method searches the tree for a word by traversing traverse it until either the node associated with the word or a null value is found returns either the isWord boolean of the node or false respectively. To delete method, search for the node in the tree, and sets isWord to false if it is found. The autocomplete method traverses every branch with a non-zero child count until it finds a valid word. It does not check every child of every node; it moves the cursor every time a child has a non-zero child count.

There are 370099 words in the English dictionary I'm using. Each hash table currently has a max size of 2000003, and each hash table double hashes with 2000001. I chose not to make this size dynamic because 2000003 items was sufficiently large for all of the applications I tested. These tables each store an object in the table. The String only hash table doesn't really need its own object, but it's more flexible if I decide to add more functionality. The add method of first hash table hashes each word and stores it in the hash table. The isWord method uses double hashing to search for the String and returns false if there is either a null, or the data attribute of the object is null (the word has been deleted). The delete method uses double hashing to search for the word in the hash table and sets the data attribute of the object to null if it finds it. Because the data is hashed and hashes cannot be inversed, it would be impossible to use hashing for an autocomplete method without storing additional data in each hash object, which the other hash table does. The autocomplete method of this hash table ignores hashing and parses the entire array searching for matching substrings in the same way as the array data structure. I tend to call this structure the normal hash, and the slightly more complex hash table the searchable hash because it's easier to search for related words.

The second hash table also stores an isWord boolean, and the number of children in each object that is hashed. The children integer refers to children in the same way is in the tree even though these objects do not actually have children; children is the count of valid words that contain this substring. Similar to adding to the tree data structure, the add method of this hash table not only adds the word to the table with isWord set to true, but also stores every substring of every word in the table. Adding uses hashing to search for every substring, if it finds is found, the child count of the object is increased. Otherwise the hashDataObject is initialized, isWord is set to false, and the count is set to one. Deleting from this hash table requires checking hashing to search for the word twice. The first just searches for the word, the second search also finds every substring of the word and decreases the child count by one, and then sets the isWord property of the word to be deleted to false. The autocomplete method of this hash table searches for the String parameter in the hash table, and while the children count is greater than 0, it proceeds in the same manner as the tree. It iterates through the alphabet adding one letter to the word. If the new String is found in the hash table and is a word the new String is returned. If the new String is not a word but it still has children, the process is repeated adding another character. The hash table ends with around 880000 values. It's not near full, but the hash table could benefit from its maximum size being increased.

2. **Experimental procedures**

To analyze the runtime of the add and method, I recorded the run time of cumulatively adding or deleting N words. The relationship between the run time complexity of a single add or delete and the cumulative run time complexity of n adds or deletes can be expressed by the Euler-Maclaurin integration formula show below (http://mathworld.wolfram.com/PowerSum.html 41)

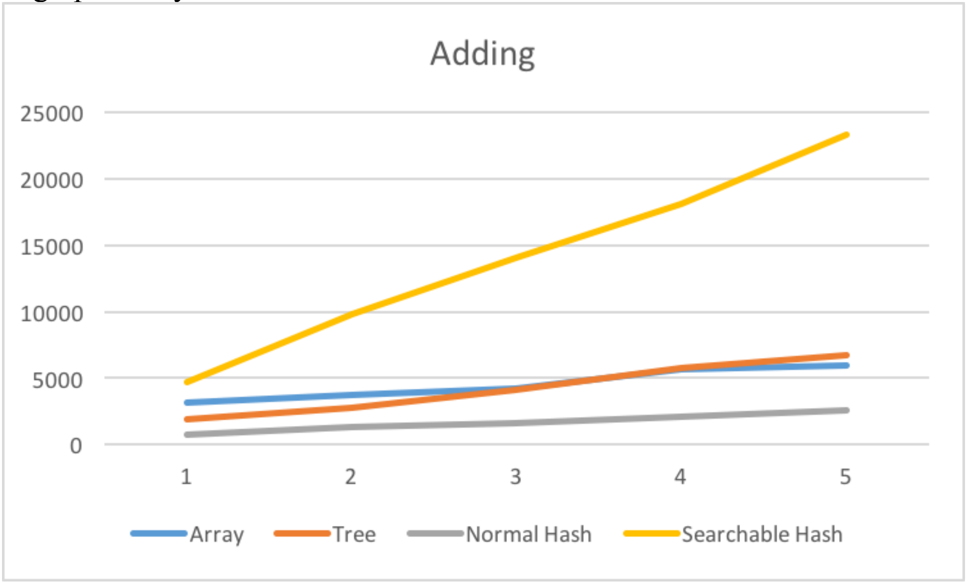
$$\sum_{k=1}^{n} f(k) = \int_{1}^{n} f(x)\,dx + \frac{1}{2} f(1) + \frac{1}{2} f(n) + \frac{1}{2!} B_2 [f'(n) - f'(1)] + \ldots$$

Where f(k) represents the runtime of add or delete. For polynomials and logarithmic functions, the most significant term of this expansion is the integral of f(x)dx. Therefore, the big O runtime of add or delete is the most significant term of the derivative of the cumulative run time of adding or deleting N words. Because adding and deleting the entire English dictionary takes around a second, to test the run time I will create some arbitrary number dictionaries of linearly spaced sizes. These data structures will contain every nth word or every 2/n words and so on in order to keep a wide distribution of alphabetical position of word. This will only add a constant factor to the run time, and will therefore not have an effect on the big O runtime of the overall trend shown.

To analyze the run time of the isWord method, I generated 3 separate lists of words. One list contains random words that are in each dictionary, one contains random strings that are not in each dictionary, and the last one contains substrings of a 16 character word that are added to each dictionary before testing. In each of these tests, isWord is called some arbitrary number of times to make the run time long enough to be analyzed. I tried to use words and strings of random length and alphabetization in order to lower the effects of length and position on the runtime of isWord in these data structures. I expect position to have an influence in the array, and length to have an influence in the tree. I did these three tests because I expect different behavior between these data structures when searching for words that can be found, words that cannot be found, and words of different length.

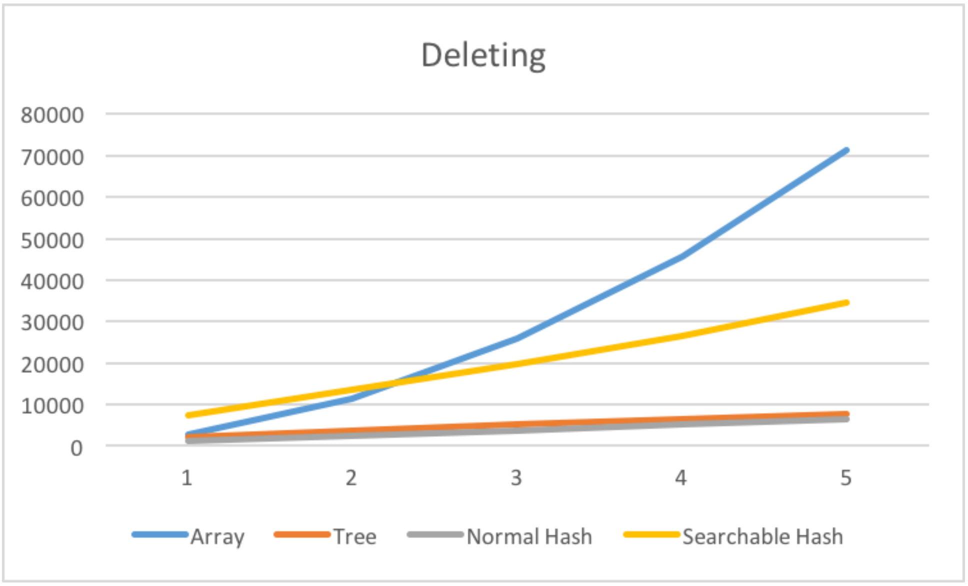
To analyze the run time of autocomplete, I generated 2 lists of strings. The first list contains strings that can be autocompleted and return a word, the second list contains strings that do not complete to any words. Autocomplete will be called on each list some arbitrary number of times in order to make the run time realistically analyzable. Again, the list of strings has a wide range of alphabetic position and length in order to lower their effects on run time.

3. **Results of Experiments**
Adding:
A graph of my results is shown below as well as data tables attached in the zip file.



All of these appear to be linear. The linear correlation coefficients for the array, tree, normal hash, and searchable hash are 0.975, 0.994, 0.991, and 0.998 respectively. Other than the array these are all very good fits. These are also all better than the logarithmic correlation coefficients, so it can reasonably be concluded that all of these functions are linear. Because this data represents the cumulative runtime, the run time of add can be expressed by its derivative. The add method for all of these data appears to be O(1).

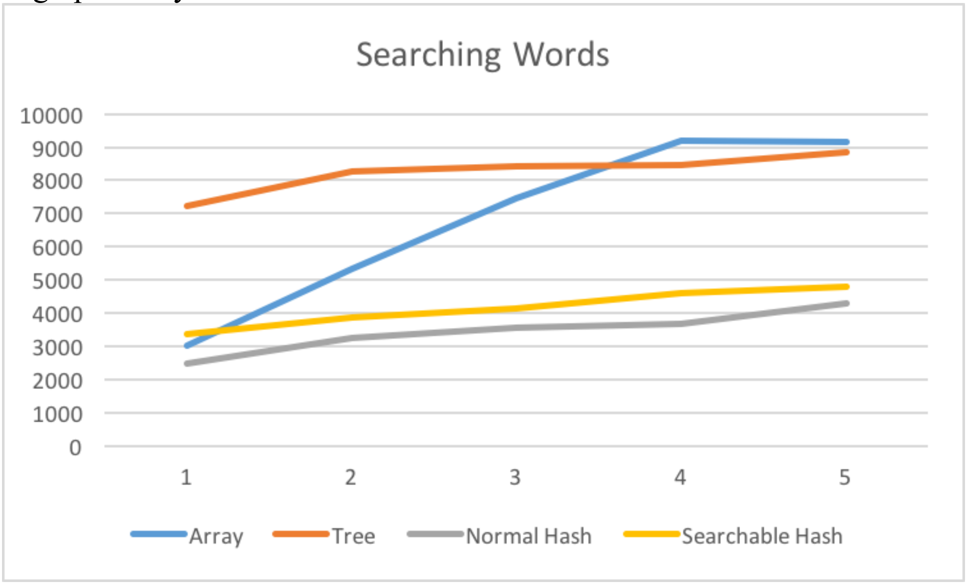
Deleting:
A graph of my results is shown below as well as data tables attached in the zip file.



The array data appears to be quadratic, and the rest of the data structures appear to have linear data. Again, they all have pretty good correlation constants. The linear correlation constants for the tree, normal hash, and searchable hash are all 0.999, and the quadratic correlation constant of the array is 0.999. Again because this is cumulative, the run time complexity of the delete method is the derivative of this data. The array is then O(N), and the rest of the data structures are O(1).

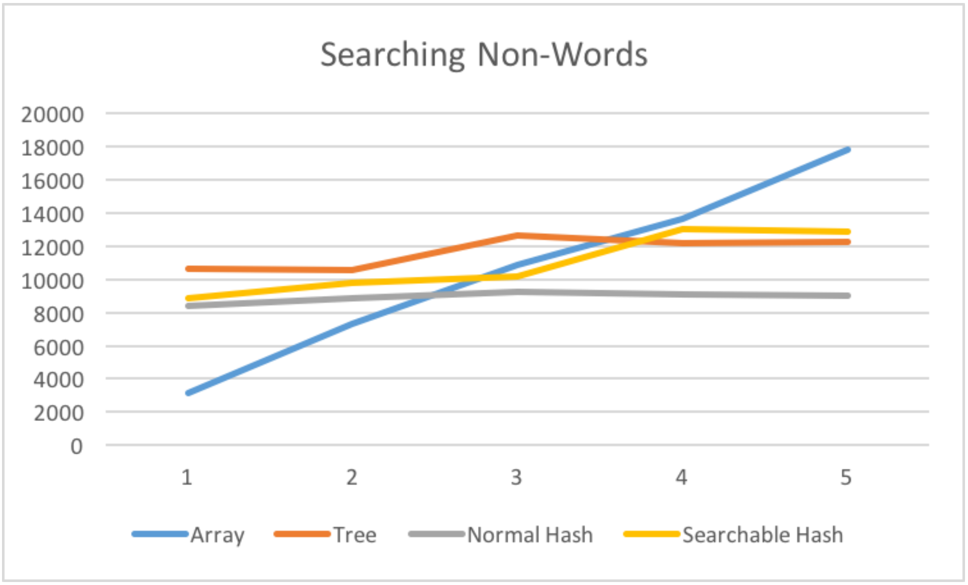
Searching:

Words:
A graph of my results is shown below as well as data tables attached in the zip file.



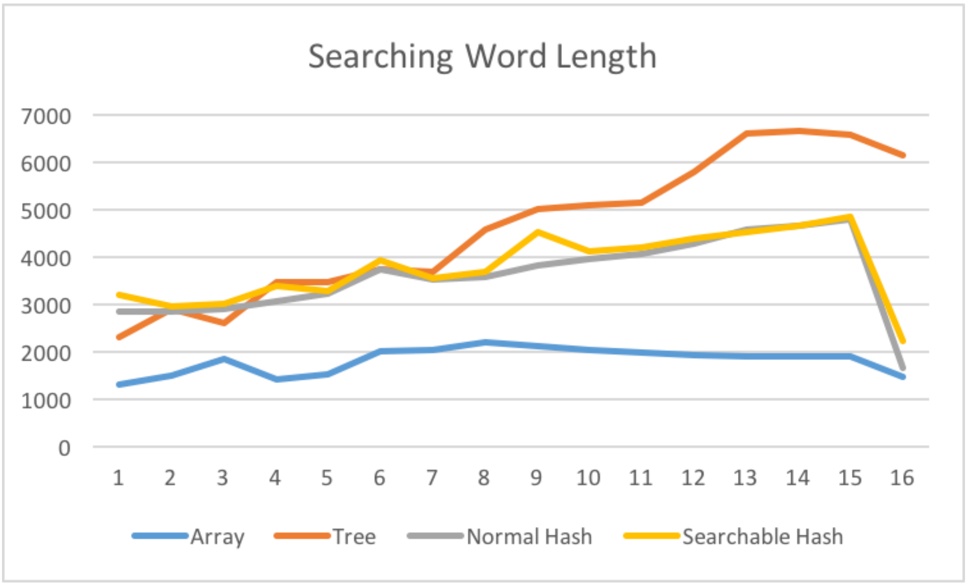
The array appears to be linear and the other three data structures appear to be linear or logarithmic. The linear correlation coefficient the array is 0.962, but ignoring the last data point it becomes 0.998. I cannot provide good justification for why the last data point doesn't follow the linear trend. I expect over repeated trials it would be linear. Both hash tables have nearly identical correlation for both linear and logarithmic fits. Each for the normal hash table is 0.97, and each for the searchable hash is 0.99. They also do not fit a constant function very well, so I cannot draw any reasonable conclusions from this data. Each fit linear and logarithmic equally well. The tree appears to be logarithmic. The tree fits none of these functions particularly well. Its linear correlation coefficient is 0.900, its logarithmic correlation coefficient is 0.958, and excluding the slight increase from N[1] to N[2] and N[4] to N[5], it appears to be constant time. Assuming the isWord is a polynomial function, the slope of a log-log plot represents the degree of the polynomial. If $y = x^n$, $\log y = n \log x$. The slope of the log of N and the log of the runtime is 0.115. This is reasonably close to 0, so a constant O(1) function is the best approximation for the tree.

Words that Are Not in Each Data Structure:
A graph of my results is shown below as well as data tables attached in the zip file.



Searching the array appears to be a linear function. Searching the tree and searchable hash appears to be constant but with a strange step in in the middle. Searching the normal hash table appears to be constant. The linear correlation coefficient for the array data is 0.998, so it very likely O(N). Again, none of the other three data structures fit a linear function or a logarithmic function particularly well. The slope of a log-log plot for the tree and normal hash table data is reasonably close to 0, 0.109 and 0.050 respectively, so searching these can reasonably be expected to be O(1). The slope of the log-log plot for the searchable hash table is 0.250, so it does not fit a constant function extremely well. The best fit is linear, with a correlation coefficient of 0.934. This is also not extremely good, so I can draw no conclusions about the run time of searching searchable hash table.

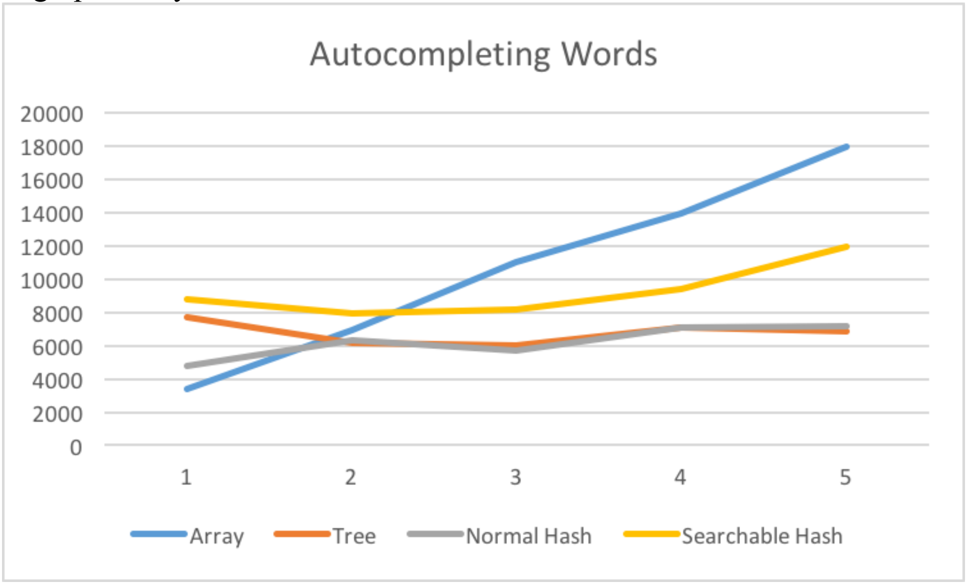
Searching Dependence on Word Length:
A graph of my results is shown below as well as data tables attached in the zip file.



None of these trends are particularly clear, especially the significant decrease in run time moving from 15 to 16 characters. The linear correlation coefficient for the array and both hash tables is below 0.5, and the slope of their log-log plots is below 0.15, so they do not appear to have any dependence on word length. The linear correlation coefficient of the tree data is 0.974, but the slope of the log-log plot is 0.415. There is some positive correlation between word length and searching the tree, but it is not clear if it is linear or some other function.

Autocompleting Partial Words

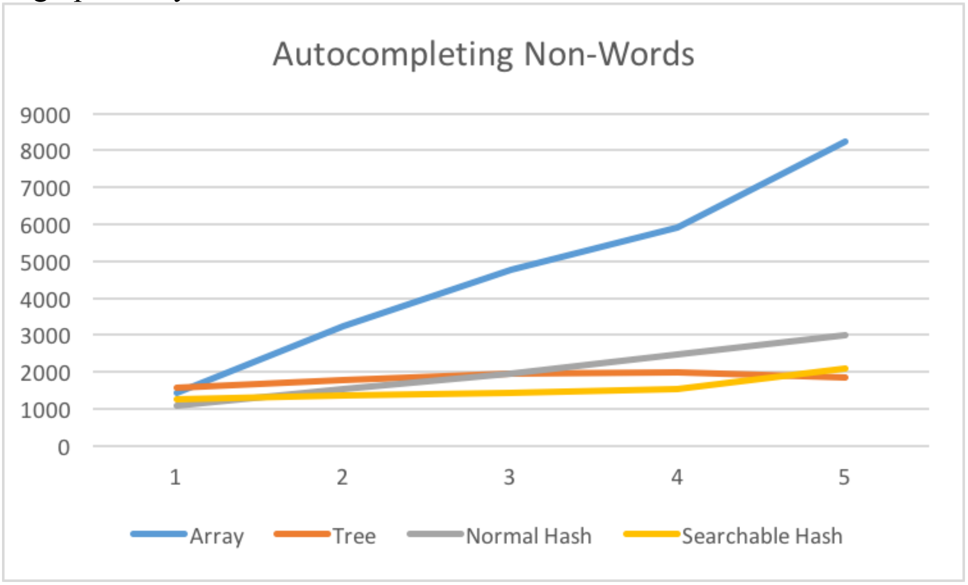
A graph of my results is shown below as well as data tables attached in the zip file.



The array data appears to be a linear function, the correlation coefficient is 0.999. Both the tree and searchable hash run times appear to decrease at first and then increase again. The may be an effect of the way the data structures are built. Because each follows the alphabetically first path, if a new word is added to the data structure that contains the partial string being searched, is shorter than the word found in the N =1 iteration, and alphabetically earlier than the word found in the previous iteration. Similarly, if a longer and alphabetically earlier word is added then the run time is expected to increase. I tried to minimize the effect by using random partial strings, but it is possible for the situation described above to occur multiple times. The slope of the log-log plots for the tree and searchable hash are -0.054 and 0.150 respectively, it can reasonably be assumed that autocompleting for the tree and searchable hash is a constant function. No conclusions can be drawn from the normal hash table data. Its linear correlation coefficient is 0.869, its logarithmic correlation coefficient is 0.889, and the slop of its log-log plot is 0.235.

Autocompleting Partial Strings that Cannot Be Completed:

A graph of my results is shown below as well as data tables attached in the zip file.



The array data and normal hash table data appear to be linear. Their correlation coefficients are 0.995 and 0.999 respectively. The slope of the log-log plot the tree data is 0.130, so autocompleting on the tree is likely an O(1) function. The searchable hash doesn't fit a linear or logarithmic function very well, and the slop of its log-log plot is 0.252. I cannot draw any significant conclusions from this data.

4.1 Theoretical Expectations

Array:

Adding to the array is an O(1) function. It simply involves writing the string passed to the method to the next index of the array. If the max size is reached, the array must be copied to an array with twice the capacity. This would not happen often, but when it does adding would be an O(N) function for N items in the array.

Searching the array is an O(N) function. This method must iterate through the array of N items, so it is O(N). Assuming that testing strings for equality is an O(1) function, the method is O(N). An argument could be made that testing for equality depends on word length and this method is O(Nw), but the N is definitely much more significant.

Deleting from the array involves searching for the word in the array, and then shifting the rest of the array to the left so there is no null element. Searching must iterate through i elements, and then N – i elements must be shifted. Shifting and iterating are both O(1). O(N-i) + O(i) is simply O(N).

Autocompleting a partial word in the array is also an O(N) function. It is identical to searching for a word, but instead of comparing the entire word, it tests if the array[i].substring(0, w) is equal to the string passed to the method.

Tree:

Adding to the tree should be independent of the size of the tree N, but it should be linearly dependent on the length of the word w. To add to the tree, the cursor starts at the root and traverses to every ith child where i is a number 0-25 corresponding the alphabetic position of the next letter in the word being added. When the cursor is at every node before the node associated with the complete string, it must either initialize the node or add one to that node's child count. Once it reaches the node associated with the string parameter, it sets the node's isWord property to true. Each of these is an O(1) function. Converting a character to its integer is simply subtracting 97 from the char value, so it is also O(1). Because a node's children are stored in an array, accessing the ith child and moving the cursor to that child is O(1). This process has to be repeated for every letter of the word, so the overall run time of adding to the tree is O(w) * O(1) + O(1) = O(w) where w is the length of the string being added.

Searching the tree should be worst case an O(w) function. On every isWord call, the tree must be traversed until either the cursor reaches the node corresponding to the word parameter, or a null or node with 0 children is reached. Traversing the tree in this search is nearly identical to traversing the tree in the adding method with the only difference being that the child counts are not changed, and if a null node is reached the function returns false rather than initializing that node. If no null node or node with 0 children is reached meaning the string passed is a word, w nodes will be traversed. If the String passed is not a word, there is some dependence on the size of the tree N, but this cannot be expressed as a function. If searching for the string "appl", the size of the tree will have no effect if there are millions of strings beginning with 'z' added to the tree. However, if some string such as "apple" is added to the tree, searching for "appl" would require traversing 4 nodes rather than the minimum of only the root node if there have been no words beginning with 'a' added to the tree. As N becomes larger, it becomes more likely that calling isWord for a string that is not a word will require traversing w nodes. Calling isWord and generally searching the tree is therefore worst case an O(w) function and best case O(1), with the average probably being closer to O(w).

Deleting from the tree first calls isWord passing whatever string is passed to delete to ensure that there is actually a word to delete. If the string passed is a word, then delete will traverse the tree subtracting one from the child count of every cursor node, and setting the isWord property of the final node to false. This traversal is identical to the traversal for adding to the tree, so the run time of delete is O(w) for isWord + (O(w) * O(1) + O(1)) for traversing the tree and setting the isWord property to false. This simplifies to O(w).

Autocompleting a partial word should also be relatively independent of the size N of the tree. Similar to searching, the total number of words in the tree does not affect the run time of the method, but certain categories of words could make the run time shorter or longer. The autocomplete first finds the node corresponding to the string passed to the method, which is an O(w) function. If this node has a child count greater than one, the cursor will move to the alphabetically next node with children. This is worst case checking the childcount property of 26 children. This will be repeated until the cursor is a word, so the number of nodes traversed is dependent on the length of the word the method returned. Like searching, if autoComplete("appl") is called and there are no words in the tree, it is an O(1) method. Worst case, autocomplete would require O(w) for searching for the parameter + O(26(l-w)) for searching through the children of the partial word. This simplifies to O(l) where l is the length of the word being returned.

Normal Hash Table:

Adding to the hash table is best case O(1) as discussed in class. For simplicity, I will assume that hashing a string is independent of the string's length. It may have some effect, but it is likely insignificant. Assuming that the previously added words are randomly distributed throughout the hash table, each index has an N/2000003 probability of resulting in a collision, where 2000003 is the static maximum size of the hash table. This somewhat unreasonably assumes that the hashes are completely random, but this probability should provide a decent approximation. (2000003-N)/2000003 is the probability that there are no collisions at any point. The expected number of indexes tried for each add is then 2000003/(2000003-N) (http://www.cut-the-knot.org/Probability/LengthToFirstSuccess.shtml). Calculating the next index to attempt using double hashing is an O(1) function, so the average run time of this method is the expected number of indexes attempted. 2000003/(2000003 -N) = (2000003 – N)/(2000003 – N) + N/(2000003 – N) = 1 + N/(2000003 - N). This cannot be simplified for some large N, because N > 2000003 results in a negative number. This function is constant because it has an upper bound of N = 2000002, but it's nonlinear so I'd rather not simplify. The average run time is O(N/(2000003 – N)), the best case is O(1) and the worst case is O(N).

Searching the hash table should have very similar run time to adding to the hash table. If searching for a string that is not in the hash table, new indexes will be attempted until an empty index is found. This is identical to adding, so the average case is O(N/(2000003 – N)). If the word is in the hash table, it is very difficult to describe the average case. The expected number of collisions is dependent on the size of the table N_i when the word was added to the hash table. The expected number of collisions when searching for a word that is in the hash table is the expected number of collisions when adding that word to the hash table, $N_i/(2000003 – N_i)$. Searching and calling isWord on the normal hash table is best case O(1), worst case O(N), average case O(N/(2000003 – N)) if the word is not in the hash table, and average case $N_i/(2000003 – N_i)$ if the word is in the hash table and N_i is the size of the hash table when the word was added.

Deleting from the hash table involves searching then setting the data property to null if the String is found. Setting to null is O(1), so its run time is then the same is searching the hash table. Best case O(1), worst case O(N), and average cases O(N/(2000003 – N)) if the word is not in the hash table and $N_i/(2000003 – N_i)$ if the word is in the hash table and N_i is the size of the hash table when the word was added.

Calling autocomplete on this hash table could not be done with hashing other than simply trying every possible string that begins with the parameter. The hash table is still an array, so this function linearly searches the hash table. Because the hash table has a set size of 2000003, this function has an upper bound and is technically an O(1) function. For a hash table with dynamic sizing, it would be O(f(N)) where f(x) is whatever function is used to define the maximum size of the hash table.

Searchable Hash Table:

Although I did not consider this while testing this data structure because it would have been impractical, the number of objects N in stored in this hash table is not the number of words stored. In this analysis, N will refer to the total number of objects rather than the total number of words.

Adding to this hash table requires adding every substring from 0 to w of the word being added to the hash table. Each add is the same as for the other hash table, so the best case would be O(w), and the worst case would be the summation of N + i – 1 from i = 1 to i = w. This simplifies to $O(w^2 + wN)$. W is most likely insignificant compared to N, so the worst case is still O(N). The average case is the summation from i =1 to i = w of (N + i -1)/(2000003 – N – i + 1). This cannot be simplified, but assuming w is relatively small and N is not near 2000003, this summation is approximately wN/(2000003 – N). The average case of adding is then O(N/(2000003-N)), the best case is O(w), and the worst case is O(N).

Substrings from 0 to w don't have to be considered during searching, so searching this hash table is identical to searching the simpler hash table. The best case is O(1), the worst case is O(N) and the average cases are O(N/(2000003 – N)) if the word is not in the hash table and $N_i/(2000003 – N_i)$ if the word is in the hash table and N_i is the size of the hash table when the word was added.

Deleting involves searching for the word, and, if it is found, setting its data property to null and decrementing the child count of every substring from 0 to w. If a word is not in this hash table, its run time is identical to searching for the word in the isWord method. If the word is in the hash table, w strings must be searched and their child counts must be decreased. Like with adding, this could be expressed as a precise summation, but the most significant terms will not change outside of the best case. The best case is O(w) for searching w strings, the average case is O(N/(2000003-N)), and the worst case is O(N).

The autocomplete method is very similar to the autocomplete method of the tree. It finds the word, checks if it has children, then tests adding characters until it a new string with children. Finding the initial string parameter has the same run time complexity as searching described above. Similar to the tree, the number of searches following the initial search depends on the length of the string that this method returns. The worst case after finding the parameter would involve 26(l-w) searches, the average case 13(l-w) searches, and the best l-w searches. These only differ by constants, so there are effectively l-w searches being done in addition to searching for the initial parameter. L and w are insignificant compared to N, so this repeated searching only affects the best case run time. The best case for l-w searches is O(l-w), the average case is O(N_i/(2000003 – N_i), and the worst case is O(N).

4.2: **Overall Conclusions**

Adding:
My testing showed that adding to all of these data structures was O(1). The matches the theoretical results. Adding to the array is expected to be O(1). Adding to the tree is expected to be O(w), but my tests did not test words of varying length. The results then should have been O(1), because w is constant with respect to N. Adding to each hash table was expected to have an average case of O(N/(2000003-N)). For N < 1000002, this expression is essentially 1; it only becomes nonlinear closer to 2000003. Neither hash had far over 1000000 objects, so the trend should have appeared to be O(1).

Deleting:
My testing showed that deleting from the array was an O(N) function, and deleting from the rest of the data structures was an O(1) function. This again matches the theoretical results. The array was expected to be O(N), the tree was expected to be O(w) which is O(1) with respect to N, and both the hash tables were expected to be O(N_i/(2000003-N_i)) where N_i represents the number of objects in the hash table when the string was added. Again, because N is much lower than 2000003, this function is nearly identical to O(1).

Searching:
My testing showed that searching should reasonably be O(w) for the tree, O(N) for the array, O(1) for Strings not in each hash table, and inconclusive for strings that were in each hash table. This matches my expectations reasonably well. O(N) is the expected run time of searching the array, and O(w) is the expected run time of searching the tree. The data for searching each hash table was not as close to O(1) as I would expect, but the theoretical and experimental

values do not disagree. Because N does not approach 2000003, the average case of searching each hash table would approximate O(1).

Autocomplete:
My testing showed that autocompleting was O(N) for the array, O(1) for the tree, and O(1) but mostly inconclusive for the searchable hash table, and O(N) but mostly inconclusive for the normal hash table. This also matches the expected values. The array was expected to be O(N), and the tree was expected to be O(l) where l is the length of the word being returned, which is O(1) with respect to N. The searchable hash table was expected to be have $O(N_i/(2000003 - N_i)$ run time, but within my range of N values this would appear to O(1). I would have liked my data to be more clearly O(1), but it supports the theoretical value.
The normal hash table should have somewhat strange run times for autocompleting words that could be found because it linearly searches the hash table for them. Their positions are essentially random, so collisions could cause somewhat significant but unpredictable changes an N changes. This may explain the lack of a trend in run time for words that can be found. For words that are not in the hash table, the function was expected to be O(1) rather than O(N) as the data suggests. This may be due to the fact that testing equality of two Strings is slower than testing if an object is null. N string comparisons must be done, and 2000003 – N null comparisons must be done.

Conclusions:
The tree and searchable hash table both essentially have O(1) run time for all of these methods. In terms of actual run time, the tree was faster for adding, deleting, and autocompleting, but the hash table was faster for searching. In practice the searchable hash table did not perfectly math the expected run time; it would be interesting to see if it nicely fits O(1) over more trials or over higher n values and a dynamically sized hash table.

5. Appendix
program files:
dataGraphs.xlsx: An excel file containing all of my data and statistical analysis.
Dict.txt: A text file containing an English dictionary with one word per line. It's mostly alphabetized but not entirely.
Dict.java: an interface for these data structures.
DictParser.java: the file with my main method. I do all of the run time analysis here.
CharTreeNode.java: the class for the CharTreeNode object to be used in the tree.
CharTree.java: Implements Dict, the tree data structure and associated methods.
DictList.java: Implements Dict, the array data structure and associated methods.
StringHashObject.java: The object being stored in the simpler/normal hash table.
StringHash.java: Implements Dict, the normal hash table and associated methods.
SearchHashObject.java: The object being stored in the more searchable hash table.
SearchHash.java: Implements Dict, the more searchable hash table and associated methods.